SpotLess: Concurrent Rotational Consensus Made Practical through Rapid View Synchronization

Dakai Kang, Sajjad Rahnama, Jelle Hellings†, Mohammad Sadoghi
Exploratory Systems Lab, Department of Computer Science, University of California, Davis
†Department of Computing and Software, McMaster University

***Abstract*—The emergence of blockchain technology has renewed the interest in *consensus-based* data management systems that are resilient to failures. To maximize the throughput of these systems, we have recently seen several prototype consensus solutions that optimize for throughput at the expense of overall implementation complexity, high costs, and reliability. Due to this, it remains unclear how these prototypes will perform in real-world environments.**

In this paper, we present SPOTLESS, a novel concurrent rotational consensus protocol made practical. Central to SPOTLESS is the combination of (1) a *chained rotational consensus design* for replicating requests with a reduced message cost and low-cost failure recovery that eliminates the traditional complex, error-prone view-change protocol; (2) the novel *Rapid View Synchronization* protocol that enables SPOTLESS to work in more general network assumptions, without a need for a *Global Synchronization Time* to synchronize view, and recover valid earlier views with the aid of non-faulty replicas without the need to rely on the primary; (3) a high-performance *concurrent consensus architecture* in which independent instances of the chained consensus operate concurrently to process requests with high throughput, thereby avoiding the bottlenecks seen in other rotational protocols.

Due to the concurrent consensus architecture, SPOTLESS greatly outperforms traditional primary-backup consensus protocols such as PBFT (by up to 430%), NARWHAL-HS (by up to 137%), and HOTSTUFF (by up to 3803%). Due to its reduced message cost, SPOTLESS is even able to outperform RCC, a state-of-the-art high-throughput concurrent consensus protocol, by up to 23%. Furthermore, SPOTLESS is able to maintain a stable and low latency and consistently high throughput even during failures.

I. INTRODUCTION

The emergence of BITCOIN [1] and blockchain technology has renewed the interest in *consensus-based* resilient data management systems (RDMSs) [2]–[7] that can provide resilience to failures and can manage data between fully-independent parties (federated data management). Due to these qualities, there is widespread interest in RDMSs with applications in finance, health care, IoT, agriculture, fraud-prevention, and other industries [8]–[12].

Although BITCOIN builds on many pre-existing techniques, the novel way in which BITCOIN used these techniques was a major breakthrough for resilient systems, as BITCOIN showed that resilient systems with thousands of participants can solve large-scale problems [2], [13]. Furthermore, BITCOIN did so in a *permissionless* way without requiring a known set of participants and allowing participants to join and leave the system at any time. The highly-flexible permissionless design of blockchains such as BITCOIN and ETHEREUM [14] is not suitable for high-performance RDMSs, however: their abysmal transaction throughput, high operational costs, and per-transaction costs make them unsuitable for typical data-based applications [1], [2], [13]–[17]. Instead, data-based applications often are deployed in an environment with a set of identifiable *participants* (who may behave arbitrarily) due to which they can use *permissioned* designs using primary-backup consensus protocols [3], [18]–[22] such as the *Practical Byzantine Fault Tolerance consensus protocol* (PBFT) [3].

RDMSs with fine-tuned primary-backup consensus implementations can process hundreds-of-thousands client requests per second [25]. Such high-throughput implementations come with severe limitations, however. First, in primary-backup consensus, a single replica (the primary) coordinates the replication of requests. Due to this central role of the primary, performance is usually bottlenecked by the network bandwidth or computational resources available to that primary [23], [26], [27]. Furthermore, the central role of the primary is detrimental to scalability, due to which high throughput can only be achieved on small-scale deployments. Finally, the techniques necessary in primary-backup consensus to reach high throughput (e.g., *out-of-order processing* [3], [25]) require complex implementations that keep track of many partially-processed rounds of consensus. When recovering from failures, this normal-case complexity necessitates complex and costly (in terms of message size and duration) *view-change protocols* to figure out which of these partially-processed consensus rounds can contribute to a consistent recovered state. Recently, we have seen two significant developments to address these limitations in isolation.

First, the introduction of *concurrent consensus protocols* such as RCC [23], MIRBFT [26] and ISS [27] have significantly improved the scalability and performance of high-throughput consensus. These concurrent consensus protocols do so by taking a primary-backup consensus protocol such as PBFT as their basis and then run multiple instances (each with a distinct primary, e.g., each non-faulty replica is a primary of its own instance) at the same time, this to remove any single-replica bottlenecks. On the one hand, the concurrent consensus is able to eliminate bottlenecks, improve scalability, and improve performance. On the other hand, existing concurrent consensus protocols do so by further increasing both the implementation complexity and the cost of *recovery*. For example, RCC shuts down faulty primaries for

Protocol	Environment		Concurrent	Chained	Threshold	Phases	Communication Complexity		
	Safety	Liveness	Consensus	Consensus	Signatures		Messages	(*at primary*)	(*per decision*)
SPOTLESS	Asynchronous	Partial Synchrony	yes	yes	no	6	$c(3\mathbf{n}^2)$	$c(3\mathbf{n})$	$\mathbf{n}^2$
PBFT [3]	Asynchronous	Partial Synchrony	no	no	no	3	$2\mathbf{n}^2$	$3\mathbf{n}$	$2\mathbf{n}^2$
RCC [23]	Asynchronous	Partial Synchrony	yes	no	no	3	$c(2\mathbf{n}^2)$	$c(3\mathbf{n})$	$2\mathbf{n}^2$
HOTSTUFF [24]	Asynchronous	Partial Synchrony	no	yes	yes	8	$8\mathbf{n}$	$4\mathbf{n}$	$2\mathbf{n}$

Fig. 1: Comparison of SPOTLESS with three state-of-the-art consensus protocols. Here, $\mathbf{n}$ is the number of replicas, c, $1 \leq c \leq \mathbf{n}$, is the number of concurrent instances, and the *per decision* cost is the amortized cost of a single consensus decision.

an exponentially increasing number of rounds after receiving sufficient complaints.

Second, the introduction of the *chained consensus protocol* HOTSTUFF [24] has provided a simplified and easier-to-implement consensus protocol with low communication costs. To achieve this, HOTSTUFF chains consecutive consensus decisions, which allows HOTSTUFF to overlap communication costs for consecutive consensus decisions and minimize the cost of *recovery*. HOTSTUFF uses low-cost recovery to change primaries after each consensus decision, thereby reducing the impact of any malicious primaries. Finally, HOTSTUFF uses *threshold signatures* [28] to make all communication phases *linear* in cost. The commendable simplicity and low cost of HOTSTUFF do come at the expense of performance and resilience, however. First, the rotational design of HOTSTUFF, which disables *out-of-order processing*, inherently bounds performance by *message delays* and makes HOTSTUFF incapable of fully utilizing computing and network resources, which causes the low throughput of HOTSTUFF. The negative impact of message delays is further compounded by the reliance on threshold signatures, which incur additional rounds of communication and have high computational costs. Furthermore, the low-cost design of *recovery* in HOTSTUFF reduces the resilience compared to PBFT, as HOTSTUFF relies on a black-box Pacemaker for *view synchronization*, which is essential to the liveness of rotational protocols [29], [30].

In this paper, we present SPOTLESS, the first practical consensus protocol that combines simplicity with high performance. SPOTLESS does so by combining a novel *chained rotational consensus design* that is optimized toward simplicity, resilience, low message complexity, and latency with a high-performance *concurrent consensus architecture*. Central to the chained rotational consensus design of SPOTLESS is *Rapid View Synchronization* (RVS), which provides continuous low-cost primary rotation to deal with malicious behavior. RVS enables SPOTLESS to work in more general network assumptions, without a need for a *Global Synchronization Time* to synchronize view, and to recover valid earlier views with the aid of non-faulty replicas without the need to rely on the primary.

The rotational design of SPOTLESS eliminates the need for the traditional complex and error-prone view-change protocols found in PBFT and its variants: due to the rotational design of SPOTLESS, only information on a single round is used during recovery. In addition, RVS provides strong *view synchronization*, resolving the liveness issues of previous works. Furthermore, RVS does not require costly threshold signatures and provides robust failure recovery steps even when communication is unreliable. Finally, by combining the *chained rotational consensus design* with a *concurrent consensus architecture*, we remove the bottleneck of message delays typically seen in rotational designs *without* having to resort to highly-complex implementation techniques such as out-of-order processing.

To evaluate the performance of SPOTLESS in practice, we have implemented SPOTLESS in APACHE RESILIENTDB (Incubating), our high-performance resilient blockchain database that serves as a testbed for future RDMS technology. Our evaluation shows that SPOTLESS greatly outperforms existing consensus protocols such as PBFT [3] by up to 430%, NARWHAL-HS [31] by up to 137%, and HOTSTUFF [19] by up to 3803%. Furthermore, due to the low message complexity of SPOTLESS, it is even able to outperform RCC [23] by up to 23% in normal conditions while serving client requests with lower latency in all cases. Finally, due to the robustness of RVS, SPOTLESS is able to maintain a stable latency and consistently high throughput even during failures.

Our contributions are as follows:

1) In Section III, we present the single-instance *chained consensus design* of SPOTLESS that provides the consensus replication using *rapid view synchronization*.
2) In Section IV, we provide the *concurrent consensus architecture* employed by SPOTLESS to run multiple instances of the chained consensus *in parallel*, due to which SPOTLESS has highly-scalable throughput akin to RCC.
3) In Section V, we empirically evaluate SPOTLESS in APACHE RESILIENTDB and compare its performance with state-of-the-art consensus protocols such as HOTSTUFF [19], PBFT [3], RCC [23], and NARWHAL-HS [31]. In our evaluation, we show the *excellent properties* of SPOTLESS, which is even able to achieve higher throughput than the concurrent consensus protocol RCC, while providing a low and stable latency in all cases.

In addition, we introduce the terminology and notation used throughout this paper in Section II, discuss related work in Section VI, and conclude on our findings in Section VII. Finally, we have summarized the properties of SPOTLESS and how they compare with other common and state-of-the-art consensus protocols in Figure 1.

II. PRELIMINARIES

a) System: We model our *system* as a fixed set of replicas $\mathfrak{R}$. We write $\mathbf{n} = |\mathfrak{R}|$ to denote the number of replicas and we write $\mathbf{f}$ to denote the number of *faulty replicas*. Each replica $R \in \mathfrak{R}$ has a unique identifier $\mathsf{id}(R)$ with $0 \leq \mathsf{id}(R) < \mathbf{n}$.

We assume $\mathbf{n} > 3\mathbf{f}$ (a minimal requirement to provide consensus in an asynchronous environment [25]), that non-faulty replicas behave in accordance with the protocols they are executing, and that faulty replicas can behave arbitrarily, possibly coordinated and malicious ways. We do not make any assumptions about clients: all clients can be malicious without affecting SPOTLESS.

b) Consensus: SPOTLESS is a *consensus protocol* that decides the sequence of *client requests* executed by all non-faulty replicas in the system $\mathfrak{R}$. To do so, SPOTLESS provides three *consensus guarantees* [25], [32]:

1) *Termination*. If non-faulty replica $R \in \mathfrak{R}$ decides upon an ρ-th client request, then all non-faulty replicas $Q \in \mathfrak{R}$ will decide upon an ρ-th client request;
2) *Non-Divergence*. If non-faulty replicas $R_1, R_2 \in \mathfrak{R}$ make ρ-th decisions τ_1 and τ_2, respectively, then $\tau_1 = \tau_2$ (they decide upon the same ρ-th client request).
3) *Service.* Whenever a non-faulty client c requests execution of τ, then all non-faulty replicas will eventually decide on a client request of c.

We note that we use SPOTLESS in a setting of a replicated service that executes client requests. Hence, instead of the abstract *non-triviality guarantee* typically associated with consensus [25], SPOTLESS guarantees *service*. Adapting SPOTLESS to settings where other versions of *non-triviality* are required is straightforward.

c) Communication: As consensus cannot be solved in asynchronous environments [4], we adopt the partial synchrony model of PBFT [3]: we *always* guarantee non-divergence (referred to as *safety*), while only guaranteeing termination and service during periods of reliable communication with a bounded message delay (referred to as *liveness*). We assume that periods of unreliable communication are always followed by sufficiently-long periods of synchronous communication for guaranteeing liveness.

d) Authentication: We assume *authenticated communication*, which is a minimal requirement to deal with malicious behavior: faulty replicas are able to impersonate each other but cannot impersonate non-faulty replicas. To enforce authenticated communication, we use both *message authentication codes* (MACs) and *digital signatures* (DSs) [33]. As the cheaper MACs do not guarantee *tamper-free message forwarding*, we only use MACs to authenticate messages that are not forwarded. For other messages, we use DSs. We write $(\!|v|\!)_p$ to denote a value v signed by participant p (a client or a replica). Finally, we write $\mathrm{digest}(v)$ to denote the message digest of a value v constructed using the same *secure cryptographic hash function* as the one used when signing v [33].

III. SPOTLESS DESIGN PRINCIPLES

SPOTLESS combines a chained consensus design with a high-performance concurrent architecture. To maximize resilience in practical network environments in which communication can become *unreliable* and messages can get lost, the chained consensus instances of SPOTLESS use *Rapid View Synchronization* (RVS) to assure that each instance can always recover and resume consensus.

Our presentation of individual SPOTLESS instance is broken up into five parts. Each *chained consensus instance* of SPOTLESS operates in views $v \leftarrow 0, 1, 2, 3, \ldots$. First, in Section III-A, we show the two steps in every view. Second, in Section III-B, we present the normal-case replication steps and the three-phase commit algorithm used by each chained consensus instance. Third, in Section III-C, we formalize the guarantees provided by the normal-case replication steps and prove the safety of SPOTLESS. Then, in Section III-D, we present the design of RVS. RVS bootstraps the guarantees provided by the normal-case replication toward providing per-instance consensus. Next, in Section III-E, we describe how SPOTLESS assures per-instance consensus in an asynchronous environment and formally prove the liveness of SPOTLESS.

A. Steps in Every View: Propose and Synch Primitives

View v is coordinated by the replica $\mathcal{P} \in \mathfrak{R}$ with $\mathsf{id}(\mathcal{P}) = v \bmod \mathbf{n}$. We say that $\mathcal{P}$ is the *primary* of view v and all other replicas act as *backups*. In view v, primary $\mathcal{P}$ will be able to *propose* the next client request τ upon which the system aims to achieve consensus. To do so, the system proceeds in *two steps i.e. Propose and Synch*. First, the primary inspects the existing chain and decides from which proposal it extends a new proposal, then the primary picks a valid client request τ, wraps and broadcasts a new proposal and broadcasts. Second, the backup replicas decide whether to vote for the new proposal and broadcast their decisions, where the new proposal is *conditionally prepared* by a replica if it receives $\mathbf{n}-\mathbf{f}$ concurring votes. Now, we explain the two steps in detail below.

First, primary $\mathcal{P}$ inspects the results of the preceding views to determine the highest extendable proposal $\mathbb{P}'$ past view $v-1$, such that $\mathcal{P}$ believes that at least $\mathbf{n} - \mathbf{f}$ replicas will vote for a new proposal extending $\mathbb{P}'$. We will explore how $\mathbb{P}'$ is chosen in Section III-C. Then, primary $\mathcal{P}$ picks a client request τ from some client c that it has not yet proposed and proposes τ by broadcasting a PROPOSE message of the form $\mathbb{P} := \textsc{Propose}(v, \tau, \mathrm{cert}(\mathbb{P}'))$ to all backups, in which $\mathrm{cert}(PP')$ is a *certificate* for the preceding proposal $\mathbb{P}'$ that $\mathcal{P}$ chooses. The certificate is either a list of $\mathbf{n} - \mathbf{f}$ digital signatures, and we will explain how certificates are used in Section III-C. To assure that τ cannot be forged by the primary, we assume that all client requests are digitally signed by the client c. To assure that the authenticity of $\mathbb{P}$ can be established and that $\mathbb{P}$ can be forwarded, the primary $\mathcal{P}$ will digitally sign the message $\mathbb{P}$.

Second, the backups establish whether the primary $\mathcal{P}$ correctly proposed a *unique* proposal to them. Specifically, the backups will exchange SYNC messages between them via which they can determine whether $\mathbb{P}$ is the only proposal that can collect enough endorsements to generate a certificate in the current view (necessary to provide non-divergence) and to ensure that enough non-faulty replicas received the same well-formed proposal $\mathbb{P}$ to assure that $\mathbb{P}$ can be recovered in

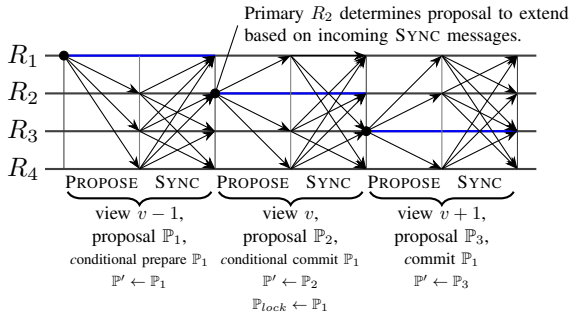

Fig. 2: A schematic representation of the normal-case replication protocol in a chained consensus instance of SPOTLESS in three consecutive views $v-1$ (primary R_1), v (primary R_2), and $v+1$ (primary R_3). $\mathbb{P}'$ refers to the *highest extendable proposal*.

any future view (independent of any malicious behavior). To do so, each backup $R \in \mathfrak{R}$ performs the following steps upon receiving message $\mathbb{P} := \text{PROPOSE}(v, \tau, \text{cert}(\mathbb{P}'))$ with digital signature $(\!|\mathbb{P}'|\!)_{\mathcal{P}}$:

1) R checks whether $(\!|\mathbb{P}'|\!)_{\mathcal{P}}$ is a *valid digital signature*;
2) R checks whether τ is a *valid client request*;
3) R checks whether view v is the *current* view; and
4) R checks whether $\text{cert}(\mathbb{P}')$ is valid if R has not *conditionally prepared* $\mathbb{P}'$.

Only if the proposal $\mathbb{P}$ passes all these checks, the replica R will consider $\mathbb{P}$ to be *well-formed*. In this case, backup R *records* $\mathbb{P}$. If $\mathbb{P}$ is the first *acceptable* proposal R receives in view v (we detail the conditions of *acceptable* proposals in Section III-C), R will broadcast the message $ms_R := \text{SYNC}(v, \text{claim}(\mathbb{P}), \mathbb{CP})$, in which $\text{claim}(\mathbb{P}) := (v, \text{digest}(\mathbb{P}), (\!|\mathbb{P}|\!)_{\mathcal{P}})$ is a *claim* that $\mathbb{P}$ is the well-formed proposal that backup R received in view v, and $\mathbb{CP}$ is a set of pairs in the form of (view, digest) for the proposals that R has *conditionally prepared*. We will explore the details of $\mathbb{CP}$ in Section III-C.

Otherwise, if backup R determines a failure in view v (R did not receive any valid proposals in view v while it should have received one), then R will end up broadcasting the message $ms_R := \text{SYNC}(v, \text{claim}(\varnothing), \mathbb{CP})$ to all backup replicas, claiming to have not received any valid proposals in view v.

To assure that the authenticity of ms_R can be established *without verifying digital signatures* and that ms_R can be forwarded, the replica R will include both a message authentication code and the digital signature $(\!|ms_R|\!)_R$. To reduce computational costs, the message authentication codes of SYNC messages are *always* verified, whereas digital signatures are *only verified in cases where recovery is necessary*, we refer to Section III-D for the exact verification rules.

B. Chained Three-Phase Commit Algorithm in Normal Case

SPOTLESS adopts a three-phase commit algorithm. Each phase takes one view, and SPOTLESS rotates the primary view by view, to eliminate complex failure detection and recovery. A successful first phase of SPOTLESS establishes a *conditional prepare* that ensures non-divergence of proposals within the view; a successful second phase of SPOTLESS establishes a *conditional commit*; and a successful third phase achieves a *commit* that ensures the preservation of proposals across views. We explain the conditions to establish these three proposal states (i.e., *conditional prepare*, *conditional commit*, and *commit*) later in Definition III.2. In Figure 2, we have visualized three *consecutive* operations of a chained rotational consensus instance in views $v-1$, v, and $v+1$ with respect to a proposal $\mathbb{P}$ and in Figure 3, we present the pseudo-code of the normal-case operations of the chained consensus.

The normal-case replication protocol of SPOTLESS only establishes a minimal guarantee on the overall state of the system that can be proven with a straightforward quorum-based argument [25]:

Theorem III.1. *Consider view v of the normal-case replication of a chained consensus instance of* SPOTLESS *and consider two replicas $R_1, R_2 \in \mathfrak{R}$. If $\mathbf{n} > 3\mathbf{f}$ and replica R_i, $i \in \{1,2\}$, receives a set of authenticated messages $\{\text{SYNC}(v, \text{claim}(\mathbb{P}_{i,Q})) \mid Q \in \mathfrak{Q}_i\}$ from a set $\mathfrak{Q}_i \subseteq \mathfrak{R}$ of $|\mathfrak{Q}_i| = \mathbf{n} - \mathbf{f}$ replicas, and all $\text{claim}(\mathbb{P}_{i,Q})$, $i \in \{1,2\}$, represent proposal $\mathbb{P}_i$, then $\mathbb{P}_1 = \mathbb{P}_2$.*

C. Rules Guaranteeing Safety

Beside the minimal guarantee in Theorem III.1, SPOTLESS ensures the safety of the system via a set of rules that non-faulty replicas must follow. In the meantime, the rules play an important role in helping restore liveness while ensuring safety. Before exploring the rules, we first introduce the necessary terminology.

Definition III.2. Let $\mathbb{P} := \text{PROPOSE}(v, \tau, \text{cert}(\mathbb{P}'))$ be a well-formed proposal in view v. We say that $\mathbb{P}'$ is the *preceding proposal* of $\mathbb{P}$. For any two proposals $\mathbb{P}_1$ and $\mathbb{P}_2$, we say that $\mathbb{P}_1$ *precedes* $\mathbb{P}_2$ if $\mathbb{P}_1$ is the preceding proposal of $\mathbb{P}_2$ or if there exists a proposal $\mathbb{P}^*$ such that $\mathbb{P}_1$ precedes $\mathbb{P}^*$ and $\mathbb{P}^*$ is the preceding proposal of $\mathbb{P}_2$. Let $\text{precedes}(\mathbb{P})$ be the set of all proposals that precede $\mathbb{P}$.

We say that a replica *records* $\mathbb{P}$ if it determines that $\mathbb{P}$ is well-formed (Line 17 in Figure 3) and say that it *accepts* $\mathbb{P}$ if it broadcasts SYNC messages with $\text{claim}(\mathbb{P})$.

We say that a replica *conditionally prepares* $\mathbb{P}$ if the replica received $\mathbb{P}$ and, during view v, the replica receives $\mathbf{n} - \mathbf{f}$ concurring votes for $\mathbb{P}$, i.e. a set of messages $\{\text{SYNC}(v, \text{claim}(\mathbb{P}_Q), \mathbb{CP}) \mid Q \in \mathfrak{Q}\}$ with $\mathbb{P}_Q = \mathbb{P}$ from a set $\mathfrak{Q} \subseteq \mathfrak{R}$ of $|\mathfrak{Q}| = \mathbf{n} - \mathbf{f}$ replicas. We say that a replica *conditionally commits* $\mathbb{P}$ if, in a future view $w > v$, the replica *conditionally prepares* a proposal of the form $\mathbb{P}' := \text{PROPOSE}(w, \tau', \text{cert}(\mathbb{P}))$ that extends $\mathbb{P}$. We say that a replica *locks* $\mathbb{P}$ if $\mathbb{P}$ is the highest proposal that it *conditionally commits*, denoted by $\mathbb{P}_{\text{lock}}$. Also, we say that a replica *commits* $\mathbb{P}$ if, in a future view $u > v$, the replica *conditionally prepares* a proposal of the form $\mathbb{P}'' := \text{PROPOSE}(u, \tau'', \text{cert}(\mathbb{P}'))$ that extends $\mathbb{P}'$, with $u = w + 1 = v + 2$. We say that two proposals are conflicting if the preceding proposals of these two proposals are disjoint.

Let $\mathcal{P} \in \mathfrak{R}$ be the replica with $v = \mathsf{id}(\mathcal{P}) \bmod \mathbf{n}$ (the primary).

function ACCEPTABLE($\mathbb{P} :=$ PROPOSE$(v, \tau, \mathrm{cert}(\mathbb{P}'))$) **do**
  Let $\mathbb{P}_{\text{lock}} :=$ PROPOSE$(v_{\text{lock}}, \tau_{\text{lock}}, \mathrm{cert}(\mathbb{P}^*))$ be the highest proposal that this replica conditionally committed.
  return R *conditionally prepared* proposal $\mathbb{P}'$ and either $v' < v_{\text{lock}}$ or $\mathbb{P}_{\text{lock}} \in (\{\mathbb{P}'\} \cup precedes(\mathbb{P}'))$.

function HIGHESTEXTENDABLE() **do**
  for v from *CurrentView* down to 0 **do**
    if $\mathcal{P}$ *conditionally prepared* proposal $\mathbb{P}'$ of view v **then**
      if $\mathcal{P}$ has a valid $\mathrm{cert}(\mathbb{P}')$ **then**
        return $\mathbb{P}', \mathrm{cert}(\mathbb{P}')$.
      else if $\mathcal{P}$ receives SYNC$(v_i, \mathrm{claim}()_i, \mathbb{CP}_i)$ from $\mathbf{n} - \mathbf{f}$ replicas R_i with $\mathrm{claim}(\mathbb{P}') \in \mathbb{CP}_i$ **then**
        return $\mathbb{P}', \mathrm{claim}(\mathbb{P}')$.

Primary role (running at the primary $\mathcal{P}$ of view v) **:**

$\mathbb{P}, \mathrm{cc} :=$ HIGHESTEXTENDABLE().
Awaits receipt of a valid client request $(\!|\tau|\!)_c$.
Broadcasts PROPOSE(v, τ, cc) to all replicas.

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

event R receives a *well-formed* proposal $\mathbb{P}$ **do**
  if R has not sent SYNC message in view v and ACCEPTABLE($\mathbb{P}$) **then**
    Broadcasts SYNC$(v, \mathrm{claim}(\mathbb{P}), \mathbb{CP})$ to all replicas.

event R determines a failure in view v **do**
  Broadcasts SYNC$(v, \mathrm{claim}(\varnothing), \mathbb{CP})$ to all replicas.

event R receives SYNC$(v, \mathrm{claim}(\mathbb{P}), \mathbb{CP})$ from $\mathbf{n} - \mathbf{f}$ replicas **do**
  Conditionally prepares $\mathbb{P}$.

event R receives SYNC$(v', \mathrm{claim}(), \mathbb{CP})$ messages with $v' > v$ and $\mathrm{claim}(\mathbb{P}) \in \mathbb{CP}$ from $\mathbf{f} + 1$ replicas **do**
  Conditionally prepares $\mathbb{P}$.

event R receives SYNC$(v, \mathrm{claim}(\mathbb{P}), \mathbb{CP})$ from $\mathbf{f} + 1$ replicas **do**
  if R has not sent SYNC message in view v **then**
    Broadcasts SYNC$(v, \mathrm{claim}(\mathbb{P}), \mathbb{CP})$ to all replicas.
  if R does not know $\mathbb{P}$ **then**
    Send ASK$(v, \mathrm{claim}(\mathbb{P}))$ to the $\mathbf{f} + 1$ replicas.

event R receives ASK$(v, \mathrm{claim}(\mathbb{P}))$ from R' and R has *recorded* $\mathbb{P}$ **do**
  Send $\mathbb{P}$ to R'.

Fig. 3: The replication protocol in a SPOTLESS instance.

The proposal states *conditionally committed* and *committed* are analogous to the proposal states *prepared* and *committed* in traditional non-chained protocols such as PBFT [3]. In Figure 2, we show how a proposal establishes the three states in the normal case. Using this terminology, we can specify the safety guarantee that individual SPOTLESS instances will maintain on the system: *no two conflicting proposals $\mathbb{P}_i$ and $\mathbb{P}_j$ can be both committed, each by a non-faulty replica*, which we will prove in Theorem III.4.

To guarantee safety, central to the design principle are the rules that non-faulty replicas follow when deciding whether to *extend, accept*, or *conditionally prepare* a proposal.

The primary $\mathcal{P}$ can construct a *certificate* for proposal $\mathbb{P}$ after $\mathcal{P}$ *recorded* $\mathbb{P}$ and received $\mathbf{n} - \mathbf{f}$ SYNC messages with valid signatures for $\mathbb{P}$, i.e.

$$S = \{\text{SYNC}(v - 1, \mathrm{claim}(\mathbb{P}_Q), \mathbb{CP}) \mid Q \in \mathfrak{Q}\}$$

with $\mathbb{P}_Q = \mathbb{P}$ and a valid signature from a set $\mathfrak{Q} \subseteq \mathfrak{R}$ of $|\mathfrak{Q}| = \mathbf{n} - \mathbf{f}$ replicas. The set S will be used to construct the certificate. Even if R fails to receive sufficient SYNC messages to *conditionally prepare* $\mathbb{P}$ in view $v - 1$, R will *conditionally prepare* $\mathbb{P}$ if it receives a valid *certificate* $\mathrm{cert}(\mathbb{P})$.

Each SYNC message includes $\mathbb{CP}$ that consists of the views and digests of the sender's $\mathbb{P}_{\text{lock}}$ and all *conditionally prepared* proposals with a higher view than the view v_{lock} of proposal $\mathbb{P}_{\text{lock}}$:

$$\mathbb{CP} := \{v_{\mathbb{P}}, \mathrm{digest}(\mathbb{P}) \mid \mathbb{P} \text{ is conditionally prepared} \wedge v_{\text{lock}} \leq v_{\mathbb{P}}\}.$$

R *conditionally prepares* $\mathbb{P}$ if it receives a set $S'$of SYNC messages from $\mathbf{f} + 1$ replicas claiming to have *conditionally prepared* $\mathbb{P}$, which implies at least one non-faulty replicas have *conditionally prepared* $\mathbb{P}$ after receiving $\mathbf{n}-\mathbf{f}$ concurring votes, where

$$S' = \{\text{SYNC}(w_{Q'}, \mathrm{claim}(\mathbb{P}_{Q'}), \mathbb{CP}_{Q'}) \mid Q' \in \mathfrak{Q}'\}$$

with $w_{Q'} \geq v$, $\mathbb{P} \in \mathbb{CP}_{Q'}$, and $\mathfrak{Q}' \subseteq \mathfrak{R}$ with $|\mathfrak{Q}'| = \mathbf{f} + 1$.

A non-faulty primary of view v considers a proposal $\mathbb{P}'$ to be *extendable* if either of the following conditions is met:

- E1 $\mathcal{P}$ has a valid *certificate* for $\mathbb{P}'$;
- E2 $\mathcal{P}$ has received a set of SYNC messages from $\mathbf{n} - \mathbf{f}$ replicas that claim to have *conditionally prepared* $\mathbb{P}'$, i.e.

$$\{\text{SYNC}(w_Q, \mathrm{claim}(\mathbb{P}_Q), \mathbb{CP}_Q) \mid Q \in \mathfrak{Q}\}$$

with $w_Q < v$ and $\mathbb{P}' \in \mathbb{CP}_{Q'}$ and $\mathfrak{Q} \subseteq \mathfrak{R}$ with $|\mathfrak{Q}| = \mathbf{n}-\mathbf{f}$.

The primary $\mathcal{P}$ backtracks to earlier views to find the highest *extendable* proposal $\mathbb{P}'$ and then sets the preceding proposal to $\mathbb{P}'$. If $\mathbb{P}'$ satisfies E1, then $\mathcal{P}$ broadcasts $\mathbb{P} :=$ PROPOSE$(v, \tau, \mathrm{cert}(\mathbb{P}'))$. Otherwise, $\mathcal{P}$ broadcasts $\mathbb{P} :=$ PROPOSE$(v, \tau, \mathrm{claim}(\mathbb{P}'))$.

When receiving a well-formed new proposal of the form $\mathbb{P} :=$ PROPOSE$(v, \tau, \mathrm{cert}(\mathbb{P}'))$ or PROPOSE$(v, \tau, \mathrm{claim}(\mathbb{P}'))$, a replica R determines whether to *accept* $\mathbb{P}$ based on the following rules:

- A1 Validity Rule: R has *conditionally prepared* $\mathbb{P}'$.
- A2 Safety Rule: $\mathbb{P}'$ extends R's locked proposal $\mathbb{P}_{\text{lock}}$, i.e. $\mathbb{P}_{\text{lock}} \in (\{\mathbb{P}'\} \cup \mathrm{precedes}(\mathbb{P}'))$.
- A3 Liveness Rule: $\mathbb{P}'$ has a higher view than $\mathbb{P}_{\text{lock}}$.

If A1 holds and either A2 or A3 holds, then R broadcasts SYNC$(v, \mathrm{claim}(\mathbb{P}), \mathbb{CP})$. Otherwise, R keeps waiting for a proposal satisfying the acceptance requirement until its timer expires.

Due to unreliable communication or faulty behavior, non-faulty replica R may fail to receive any *acceptable* proposal from primary $\mathcal{P}_v$ but receive a set M' consisting of $\mathbf{f} + 1$ SYNC messages with the same $\mathrm{claim}(\mathbb{P})$, formally, R receives

$$M' = \{\text{SYNC}(v, \mathrm{claim}(\mathbb{P}_{Q^M}), \mathbb{CP}_{Q^m}) \mid Q^M \in \mathfrak{Q}^{\mathfrak{M}}\}$$

with $\mathbb{P}_{Q^M} = \mathbb{P}$ from a set $\mathfrak{Q}^{\mathfrak{M}} \subseteq \mathfrak{R}$ of $|\mathfrak{Q}^{\mathfrak{M}}| = \mathbf{f}+1$ replicas. For easier restoration of liveness, SPOTLESS allows R to broadcast $\text{SYNC}(v, \text{claim}(\mathbb{P}), \mathbb{CP})$ if R considers $\mathbb{P}$ as *acceptable*.

In such a case, R is unaware of the full information of $\mathbb{P}$ and needs to catch up. To do so, R sends $\mathbf{a} := \text{ASK}(v, \text{claim}(\mathbb{P}))$ to the $\mathbf{f}+1$ replicas in $\mathfrak{Q}^{\mathfrak{M}}$. After a good replica $R' \in \mathfrak{Q}^{\mathfrak{M}}$ receives $\mathbf{a}$, the replica R' will forward $\mathbb{P}$ to R if it has *recorded* a well-formed $\mathbb{P}$. To reduce the overhead of this mechanism in practical implementations, replicas can choose to first send ASK messages to replicas they already trust (e.g., based on previous behavior).

Based on the design principles above, we can prove the safety property of SPOTLESS step by step:

Lemma III.3. *If a non-faulty replica R conditionally prepares $\mathbb{P} = \text{PROPOSE}(v, \tau, \text{cert}(\mathbb{P}'))$, then for each proposal $\mathbb{P}^* \in \text{precedes}(\mathbb{P})$ that precedes $\mathbb{P}$, at least $\mathbf{n} - 2\mathbf{f} \geq \mathbf{f}+1$ non-faulty replicas have conditionally prepared $\mathbb{P}^*$ and sent SYNC messages with $\mathbb{P}^* \in \mathbb{CP}$.*

We refer to our technical report [34] for the complete proofs of the results presented in this paper. Using Lemma III.3, we are able to prove safety:

Theorem III.4. *No two non-faulty replicas can commit conflicting proposals $\mathbb{P}_i$ and $\mathbb{P}_j$.*

Theorem III.4 proves SPOTLESS can ensure safety if we have *three-consecutive-view* requirement[2] for committing a proposal.

D. Bootstraping Liveness with Rapid View Synchronization

Rapid View Synchronization (RVS) bootstraps the guarantees provided by normal-case replication toward providing consensus. RVS does so by dealing with asynchronous communication and by strengthening the guarantees on proposals of preceding views. In specific, the main services provided by RVS are a best-effort and quick *view synchronization* to assure that replicas end up in the same views whenever communicaiton is sufficiently reliable and *low-cost state recovery* to enable cheap primary rotation to deal with failures of previous primaries.

To enable *Rapid View Synchronization*, for each view v, a replica must go through three states one by one:

ST1 Recording: waiting for a well-formed $\mathbb{P}$ that satisfies A1 and either A2 or A3 until state timer t_R expires;

ST2 Syncing: waiting for a set of SYNC messages with view v from a set $\mathfrak{Q} \subseteq \mathfrak{R}$ with $|\mathfrak{Q}| = \mathbf{n} - \mathbf{f}$ replicas;

ST3 Certifying: waiting for a set of messages

$$S = \{\text{SYNC}(v, \text{claim}(\mathbb{P}_{Q^A}), \mathbb{CP}) \mid Q^A \in \mathfrak{Q}^{\mathfrak{A}}\}$$

with the same claimed proposal $\mathbb{P}_{Q^A}$ from a set $\mathfrak{Q}^{\mathfrak{A}} \subseteq \mathfrak{R}$ of $|\mathfrak{Q}^{\mathfrak{A}}| = \mathbf{n} - \mathbf{f}$ replicas until timer state t_A expires.

[2]We refer to our technical report [34] for the necessity of the *three-consecutive-view* requirement (over a two-consecutive-view requirement) for SPOTLESS

Note that there is no timer for Syncing (ST2) and receiving sufficient SYNC messages is the only way to proceed to Certifying (ST3) of the same view. Some replicas may fall behind due to unreliable communication, failing to receive sufficient SYNC messages while other replicas have reached higher views. To quickly synchronize views, a replica R in view v is allowed to proceed to Syncing (ST2) of view w directly after receiving a set of messages D with views higher than or equal to w:

$$D = \{\text{SYNC}(v', \text{claim}(\mathbb{P}_{Q^D}), \mathbb{CP}_{Q^d}) \mid Q^D \in \mathfrak{Q}^{\mathfrak{D}}\}$$

with $v' \geq w > v$ from a set $\mathfrak{Q}^{\mathfrak{D}} \subseteq \mathfrak{R}$ of $|\mathfrak{Q}^{\mathfrak{D}}| = \mathbf{f}+1$ replicas. Receiving such $\mathbf{f}+1$ messages implies that one non-faulty replica has moved to view w after receiving $\mathbf{n}-\mathbf{f}$ SYNC messages of view $w-1$, then R can skip to view w directly knowing that at least the majority of non-faulty replicas have observed the higher view $w-1$. To catch up, replica R broadcasts message $S_u = \text{SYNC}(u, \text{claim}(\varnothing), \mathbb{CP}, \Upsilon)$ for each view $u, v \leq u \leq w$, in which Υ is a flag that asks replicas that receive S_u to retransmit the SYNC messages to R they broadcast in view u. With such a design, in SPOTLESS, as long as network remains synchronous, replicas falling behind are capable of catching up actively and immediately, while in previous *rotational* work such as HOTSTUFF, *view synchronization* is assumed by relying on the black-box Pacemaker. In Figure 4, we present the pseudo-code of the Rapid View Synchronization part of SPOTLESS. We have the following:

Lemma III.5. *Assume reliable communication. If replica R conditionally prepares proposal $\mathbb{P}$ by receiving $\mathbf{f}+1$ SYNC messages with $\text{claim}(\mathbb{P}) \in \mathbb{CP}$, then eventually R will record and conditionally prepare all proposals in $\text{precedes}(\mathbb{P})$.*

Using Lemma III.5, we can prove:

Theorem III.6. *Let $\mathbb{P}_h$ be the highest proposal that any replica conditionally committed. All non-faulty replicas will eventually record and conditionally prepare all proposals in $\text{precedes}(\mathbb{P}_h)$, when communication becomes synchronous for sufficiently long.*

From Theorem III.6 we know that all non-faulty replicas will learn the same *conditionally committed* chain. However, the replicas may not execute several proposals on the chain until they learn full information of the proposal via the ASK-recovery mechanism detailed in Section III-C.

E. Mechanism Guaranteeing Liveness

In some cases, replica R cannot make any progress unless it receives some specific messages from other replicas:

1) R cannot switch from Syncing (ST2) to Certifying (ST3) unless it receives $\mathbf{n}-\mathbf{f}$ SYNC messages of its current view.
2) R cannot catch up to learn a path from a *conditionally prepared* proposal to the genesis proposal unless at least $\mathbf{f}+1$ other replicas reply to its SYNC messages with flag Υ, which requires the receivers to retransmit the SYNC messages that they broadcast before.

Backup role (running at each replica $R \in \mathfrak{R}$ in view v) **:**

1: **event** R enters view v **do**
2: state := *recording* (ST1).
3: Set timer t_R.

4: **event** R receives an acceptable proposal $\mathbb{P}$ **or** t_R expires **do**
5: Broadcast SYNC$(v, \text{claim}(\mathbb{P}), \mathbb{CP})$.
6: state := *Syncing* (ST2).

7: **event** R receives $\mathbf{n} - \mathbf{f}$ SYNC messages of view v **do**
8: state := *Certifying* (ST3).
9: Sets timer t_A.

10: **event** R receives $\mathbf{n} - \mathbf{f}$ SYNC$(v, \text{claim}(\mathbb{P}), \mathbb{CP})$ of the same $\mathbb{P}$ **or** t_A expires **do**
11: Enters view $v + 1$.

12: **event** R receives $\mathbf{f} + 1$ SYNC messages with views higher than or equal to w, $w > v$ **do**
13: Let v be the current view of R.
14: **for** each view $u, v \leq u \leq w$ **do**
15: Broadcasts SYNC$(u, \text{claim}(\varnothing), \mathbb{CP}, \Upsilon)$.

Fig. 4: Rapid View Synchronization in instance.

3) R cannot *record* a proposal it did not receive from the primary, unless any replica replies to its ASK message by forwarding the corresponding PROPOSE message.

However, due to unreliable communication, R may fail to receive messages, e.g., replies to SYNC messages with flag Υ or replies to ASK messages. To deal with this case, R will periodically retransmit the messages until it receives the necessary replies.

In an asynchronous environment, one cannot reliably distinguish between communication failure (e.g., due to long and unpredictable message delays) and replica failure. Hence, consensus protocols such as HOTSTUFF [19] and many others [35], [36] simply assume to be operating after a *Global Synchronization Time*, at which point all communication is bound by some message delay such that all replicas can always reliably determine in which view they operate [19]. Such a design is inflexible in the presence of true asynchronous communication, however.

Instead, SPOTLESS instances use our *Rapid View Synchronization* mechanism to allow replicas to figure out in which view they should operate. To adapt to fluctuations in message delays, SPOTLESS will adjust the timeout interval used by individual replicas to detect replica failures. As message delays in typical deployments do not often change drastically, we choose to *not* use a traditional exponential backoff mechanism [25], but instead to adjust the timeout interval of replicas R in a more moderate way. For consecutive timeouts of the same timer in consecutive views, we only increase the timeout interval by a constant ε (after each consecutive view). If a replica receives an expected message for which the timeout interval was Δ *before* 0.5Δ, then the replica reduces the timeout by half. We have the following technical result.

Lemma III.7. *Let v be the highest view reached by a non-faulty replica after communication enters a period of synchronous communication. Non-faulty primary $\mathcal{P}_w$ with $v + 2 \leq w$ is capable of finding a proposal $\mathbb{P}'$ such that all non-faulty replicas will accept a proposal $\mathbb{P}$ extending from $\mathbb{P}'$, $\mathbb{P} :=$ PROPOSE$(w, \tau, \text{claim}(\mathbb{P}'))$.*

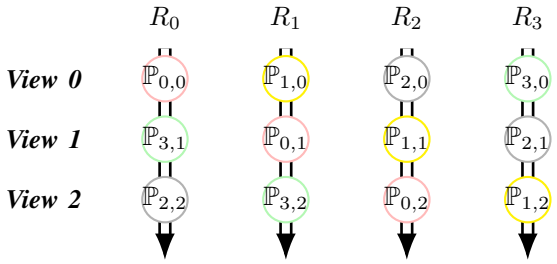

Fig. 5: Primary rotation in SPOTLESS with four replicas and four instances. The circles in the form of $\mathbb{P}_{i,v}$ on the arrow of R_r represent that R_r is the primary of instance $\mathcal{I}_i$ in view v, where $r = (i + v) \bmod \mathbf{n}$.

As a direct consequence, we have the following corollary.

Corollary III.8. Let v be the highest view reached by a non-faulty replica after communication enters a period of synchronous communication. If all non-faulty replicas have internal timers that are higher than the current maximum message delay, then the proposals of any non-faulty primary $\mathcal{P}_w$ during view w, $w \geq v + 2$, will be *conditionally prepared* after view w by all non-faulty replicas.

Corollary III.8 is at the basis of proving termination: consensus decisions are eventually made when communication is synchronous.

Theorem III.9. *All non-faulty replicas will eventually commit new proposals after communication enters a sufficiently-long period of synchronous communication.*

IV. CONCURRENT CONSENSUS

The main benefit of chained consensus, as used by SPOTLESS and HOTSTUFF [19], is that a single proposal represents the entire chain of preceding proposals. This greatly reduces the message complexity of view-changes when compared to traditional non-chained consensus protocols such as PBFT [3].

Unfortunately, chained consensus requires that consecutive consensus decisions are made one-at-a-time, thereby preventing the usage of *out-of-order processing* to maximize throughput. This makes HOTSTUFF and individual chained consensus instances of SPOTLESS significantly slower than traditional consensus protocols such as PBFT in practical deployments: primaries in PBFT can use *out-of-order processing* to propose client requests for future views while waiting on the current consensus round to finish, thereby maximizing the utilization of the network bandwidth available at the primary independent of any message delays (which dominate the time it takes to finish a single consensus round).

As an alternative to out-of-order processing, SPOTLESS will adopt concurrent consensus [23], [26]. By running multiple concurrent instances, SPOTLESS is able to effectively utilize all network bandwidth and computational resources available: when one SPOTLESS instance is waiting for a proposal to be processed (e.g., waiting for SYNC messages), other SPOTLESS instances use the available network bandwidth and computational resources to propose additional requests. We refer to

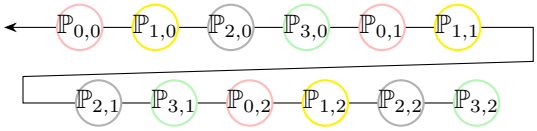

Fig. 6: Total ordering of the twelve proposals of Figure 5 made among four concurrent instances in three consecutive views.

our technical report [34] for an in-depth theoretical analysis of the benefits of concurrent processing in SpotLess. Below we illustrate the concurrent instances in SPOTLESS.

In SPOTLESS, the system runs $\mathbf{m}$, $1 \leq \mathbf{m} \leq \mathbf{n}$, SPOTLESS instances concurrently. Instances do not interfere with each other. Each instance independently deals with any malicious behavior. To enforce that each instance is coordinated by a distinct primary, the primary of $\mathcal{I}_i$ in view v is predetermined: $\mathsf{id}(\mathcal{P}_{i,v}) = (i + v) \bmod \mathbf{n}$. Figure 5 shows how SPOTLESS assigns and rotates primary in each SPOTLESS instance. As all instances rotate over all primaries, all instances are equally affected by malicious behavior. When given the choice, non-faulty replicas will prioritize instances that are in older views over other instances. Due to primary rotation and this instance prioritization, the view of all instances will remain roughly-in-sync.

Each SPOTLESS instance determines a local order of proposals. All committed proposals on the chains are totally ordered and then executed. We order the committed proposals among different instances by their view and instance identifier. We order all proposals from low view to high view and from instance $\mathcal{I}_0$ to instance $\mathcal{I}_{\mathbf{m}-1}$. Figure 6 shows the *total ordering* in SPOTLESS. Finally, each replica *executes* the committed proposals in order and informs the clients of the outcome of their requested transactions.

As individual SPOTLESS instances provide consensus and combining consensus decisions of instances is deterministic, we conclude

Theorem IV.1. *All instances of* SPOTLESS *will eventually commit new proposals after communication enters a sufficiently-long period of synchronous communication.*

V. EVALUATION

Previously, we detailed and analyzed the design of SPOTLESS, showing several theoretical advantages when compared to its peers. Next, to show the practical advantages of SPOTLESS, we will experimentally evaluate its performance, both in the normal case and during Byzantine failures. In our evaluation, we compare the performance of SPOTLESS in APACHE RESILIENTDB (Incubating), our high-performance open-source blockchain database[1], with the well-known primary-backup consensus protocols PBFT [3], HOTSTUFF [19], NARWHAL-HS [31], and our PBFT-based concurrent consensus paradigm RCC [23]. We focus on answering the following questions:

[1]the evaluation is based on release v3.0: https://github.com/resilientdb/resilientdb/releases/tag/v3.0

- Q1 Scalability: does SPOTLESS deliver on the promises to provide better scalability than other consensus protocols?
- Q2 Latency: does SPOTLESS provide low client latency while providing high throughput? What factors affect them?
- Q3 What is the impact of batching client transactions on the performance of SPOTLESS?
- Q4 How does SPOTLESS perform under Byzantine failures?
- Q5 How does *concurrent consensus* improve performance?

To study the practical performance of SPOTLESS and other consensus protocols, we implemented SPOTLESS and other protocols in APACHE RESILIENTDB. We refer to the technical report [34] for a description of APACHE RESILIENTDB and how it aids in implementing high-performance consensus. To generate experimental workloads, we used the *Yahoo Cloud Serving Benchmark* [37] provided by the Blockbench macro benchmarks [38]. In the generated workload, each client transaction queries a YCSB table with half a million active records and 90% of the transactions write and modify records. Before the experiments, each replica is initialized with an identical copy of the YCSB table. We perform all experiments on Oracle Cloud, using up to 128 machines for replicas and 32 machines for clients. Each replica and client is deployed on a `e3`-machine with a 16-core AMD EPYC 7742 processor, running at $3.4\,\text{GHz}$, and with $32\,\text{GB}$ memory.

A. The Consensus Protocols

We evaluate the performance of SPOTLESS by comparing it with a representative selection of four efficient practical consensus protocols implemented in APACHE RESILIENTDB:

PBFT *[3] and* RCC *[23].* We use an optimized out-of-order implementation of PBFT that uses message authentication codes. RCC turns PBFT into a concurrent consensus protocol.

HOTSTUFF *[19]* uses threshold signatures to minimize communication. Since existing threshold signature algorithms are expensive and quickly become the bottleneck, we use a list of $\mathbf{n}-\mathbf{f}$ `secp256k1` digital signatures to represent a threshold signature, which improves our throughput. In our experiments, we implement the pipelined CHAINED HOTSTUFF.

NARWHAL-HS *[31]* separates the replication of transactions and ordering transactions, enabling concurrent transaction dissemination. We simulate the communication complexity and computational overhead of NARWHAL-HS by running HOTSTUFF and requring replicas to broadcast messages consisting of a client batch and $2\mathbf{f} + 1$ digital signatures.

We run the concurrent protocols RCC and SPOTLESS with $\mathbf{n}$ instances unless stated otherwise.

B. Experiments

To be able to answer Questions Q1–Q5, we perform fifteen experiments in which we measure the performance of SPOTLESS and other consensus protocols. We measure *throughput* as the number of transactions that are executed per second and *latency* as the average duration between the client sending a transaction and the client receiving a response. Unless stated otherwise, all replicas are non-faulty. We run each experiment for $130\,\text{s}$ (except the seventh experiment): the first $10\,\text{s}$ are

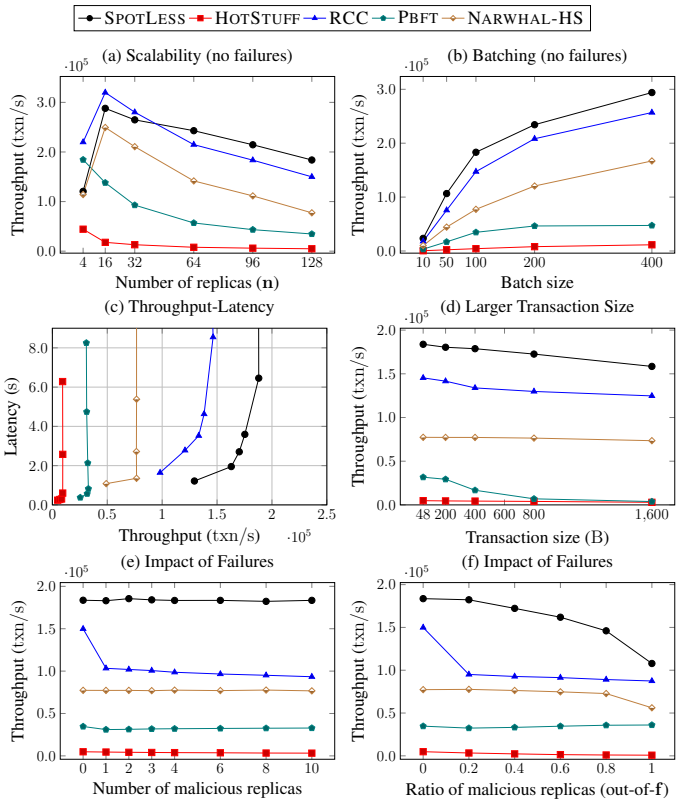

Fig. 7: Performance of SpotLess and other protocols. The number of replicas is 128 except in (a).

warm-up, and measurement results are collected over the next 120 s. We average our results over three runs.

In the *scalability experiment*, we measure throughput as a function of the number of replicas. We vary the number of replicas between $\mathbf{n} = 4$ and $\mathbf{n} = 128$ and we use a batch size of 100 txn/batch. The results are in Figure 7(a).

In the *batching experiment*, we measure the throughput as a function of the number of replicas. We use $\mathbf{n} = 128$ replicas and we vary batch size between 10 txn/batch and 400 txn/batch. The results are in Figure 7(b).

In the *throughput-latency experiment*, we measure the latency as a function of the throughput. We use $\mathbf{n} = 128$ replicas, set the batch size to 100 txn/batch, and we vary the speed by which each primary receives client requests to affect throughput and latency. The results are in Figure 7(c).

In *the transaction-size experiment*, we measure the throughput of SpotLess as a function of the individual YCSB transaction size. We use $\mathbf{n} = 128$ replicas and vary the transaction size from 48 B to 1600 B. The results are in Figure 7(d).

In the *all-throughput-failures experiment*, we measure the throughput as a function of the number of malicious replicas that do not participate in consensus. We use $\mathbf{n} = 128$ replicas, and we vary the number of faulty replicas between 0 and 10 or between 0 and $\mathbf{f}$. We make the faulty replicas non-responsive at the same time point and measure throughput afterward for 120 s. We set the timeout length in SpotLess, HotStuff, and Narwhal-HS based on the calculated average view duration and set the timeout length in RCC and Pbft based on the average client latency. The results are in Figure 7(e) and (f).

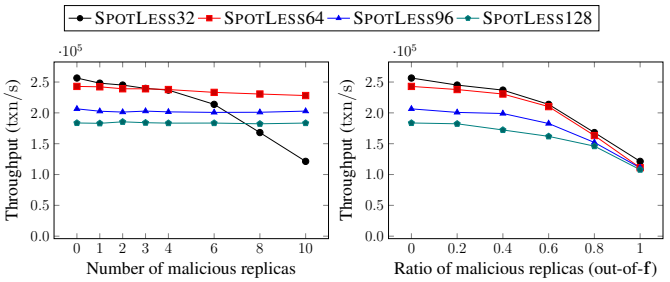

Fig. 8: Performance of SpotLess during failures as a function of the number of replicas and faulty replicas.

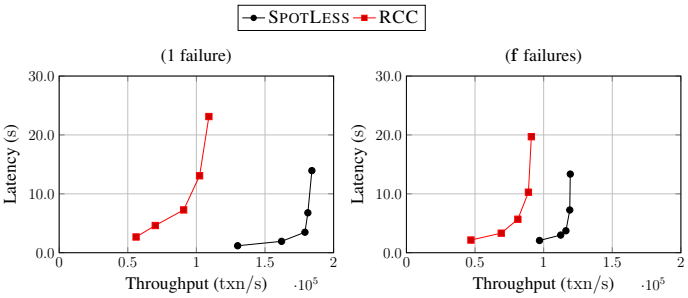

Fig. 9: System throughput-latency of SpotLess and RCC in the presence of failures (128 replicas).

In the SpotLess-*throughput-failures experiment*, we further measure the throughput of SpotLess as a function of the number of replicas and the number of malicious replicas that do not participate in consensus. We vary the number of replicas between $\mathbf{n} \in \{32, 64, 96, 128\}$ and we perform two measurements for each $\mathbf{n}$. We vary the number of faulty replicas between 0 and 10 or between 0 and $\mathbf{f}$. Based on the calculated average view duration, we have set the timeout length in SpotLess appropriately. The results are in Figure 8.

In the *throughput-latency-failure experiment*, we measure the latency of SpotLess and RCC as a function of throughput in the presence of malicious replicas. As in the throughput-latency experiment, we use $\mathbf{n} = 128$ replicas. In this experiment, we set the number of faulty, non-responsive, replicas to be 1 or $\mathbf{f}$, however. We only count the latency of proposals that are sent to non-faulty replicas. The results are in Figure 9.

In the *parallel transaction processing experiment*, we measure the performance of SpotLess and RCC as a function of the amount of *concurrent* (both protocols) and *out-of-order* (RCC) processing. To do so, we measure the throughput and latency of SpotLess and RCC as a function of the number of client batches that each primary receives. We use $\mathbf{n} = 128$ replicas and we vary the number of client batches between 12 and 200. The results are in Figure 10.

In the *throughput-Byzantine experiment*, we measure the throughput of SpotLess in the presence of attacks as a function of the number of Byzantine replicas. We consider four types of attacks:

- A1 faulty replicas are non-responsive;
- A2 faulty replicas act *malicious* when they are the primary by keeping $\mathbf{f}$ non-faulty replicas *in the dark* (by not sending proposals to them);
- A3 faulty replicas act *malicious* by sending conflicting con-

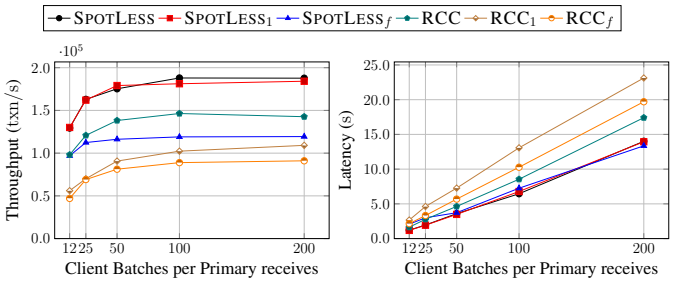

Fig. 10: Throughput and latency of SpotLess and RCC as a function of the amount of parallel transaction processing (128 replicas).

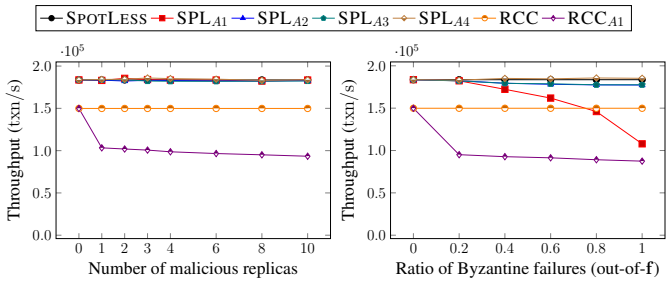

Fig. 11: Performance of SpotLess and RCC during Byzantine failures (128 replicas, four attack scenarios for SpotLess). *Note:* SPL *is the abbreviation for* SpotLess *here.*

curring votes in an attempt to cause *divergence*: they send one message to $\mathbf{f}$ non-faulty replicas and a different one to the other non-faulty replicas; and

A4 faulty replicas act *malicious* by refusing to participate in the consensus of proposals from non-faulty primaries, this in an attempt to subvert non-faulty primaries (and make them look faulty).

For comparison, we include RCC. The *victims* of these attacks are the replicas that are kept in the dark (A2), receive a proposal that is received by not more than $\mathbf{f}$ non-faulty replicas (A3), or are not responded to (A4). The throughput of RCC is not influenced by A2, A3, and A4 as long as the number of *victims* is not greater than $\mathbf{f}$. Hence, we only include the normal-case (failure-free) throughput of RCC and the throughput during A1 for comparison. We use $\mathbf{n} = 128$ replicas and we vary the number of malicious replicas between 0 and 10 or between 0 and $\mathbf{f}$. The results are in Figure 11.

In the *real-time-throughput-failure experiment*, we measure the real-time throughput of SpotLess and RCC after making malicious replicas non-responsive as a function of time. We use $\mathbf{n} = 128$ replicas, and we set the number of faulty replicas to 1 or $\mathbf{f}$. We run the experiments for 140 s, record throughput every 5 seconds, and have the failures happen at the 10th second. The results can be found in Figure 12.

In the *concurrent-consensus experiment*, we measure the throughput of SpotLess and RCC as a function of the number of replicas and concurrent instances. We use $\mathbf{n} \in \{32, 128\}$ replicas and we vary the number of concurrent instances between 1 and $\mathbf{n}$. The results can be found in Figure 13.

In the *computing-power-impact experiment*, we measure the throughput of SpotLess and other protocols as a function of the number of CPU cores in each replica. We use $\mathbf{n} = 128$ replicas and we vary the number of CPU cores between 4 and 32. The results can be found in Figure 14(a).

In the *network-bandwidth-impact experiment*, we measure the throughput of SpotLess and other protocols as a function of the bandwidth. We use $\mathbf{n} = 128$ replicas and vary the bandwidth between 500 Mbit/s and 4000 Mbit/s using FireQOS [39], a program that helps configure traffic shaping on Linux. The results can be found in Figure 14(b).

In the *global-regions experiment*, we measure the throughput of SpotLess and other protocols as a function of the number of regions. We use $\mathbf{n} = 128$ replicas and vary the number of regions between 1 and 4. For each run, the 128 replicas are uniformly distributed in the regions Oregon, North Virginia, London, and Zurich. The results can be found in Figure 14(c) and (d).

In the *non-concurrent-failure experiment*, we measure the throughput of single-instance SpotLess and HotStuff as a function of the number of faulty replicas. We use $\mathbf{n} = 128$ replicas and vary the number of malicious replicas between 0 and $\mathbf{f}$. The results can be found in Figure 15.

C. Experiment Analysis

The experimental results presented in the previous sections allows us to answer the research questions Q1–Q5. First, we observe that increasing the batch size, increases the performance of all consensus protocols, thereby answering Q3 as expected. Since the gains brought by increased the batch size are small after 100 txn/batch, we used 100 txn/batch in all other experiments.

SpotLess outperforms all other protocols in failure-free conditions (Q1, Q2). As Figure 1 shows, the amortized message complexity per decision is $\mathbf{n}^2$ for SpotLess, while it is $2\mathbf{n}^2$ for RCC. Hence, as SpotLess has fewer messages to process, SpotLess can even outperform RCC by up to 23%. Due to a *message buffer mechanism* implemented in Apache ResilientDB, which put messages in a buffer and send them together to better utilize network bandwidth, SpotLess and RCC require sufficient batches of client requests to fill the system pipeline. Otherwise, the two protocols may get stalled since no messages are sent and processed. When the pipeline is full, the latency of both SpotLess and RCC in Apache ResilientDB is dominated by the maximum throughput, as Figure 7(c), 9, and 10 show. The higher throughput is, the shorter a client request waits to be proposed and then the lower latency is. Thus, even though SpotLess needs more communication phases than RCC to commit a proposal, SpotLess has a lower latency by up to 32% than RCC. Also, SpotLess outperforms Narwhal-HS because, for each committed block, SpotLess verifies $O(n)$ MACs while Narwhal-HS verifies $O(n)$ digital signatures.

By introducing concurrent processing, SpotLess is able to outperform HotStuff, the other chained consensus protocol, by up to 3803%. In this situation, the performance of HotStuff is bottlenecked by the message delay due to the lack of out-of-order processing. From Figure 7(d) we conclude that RCC and SpotLess are able to sustain high throughput even if we increase the transaction size to 1600 B per YCSB transaction, whereas the throughput of Pbft and

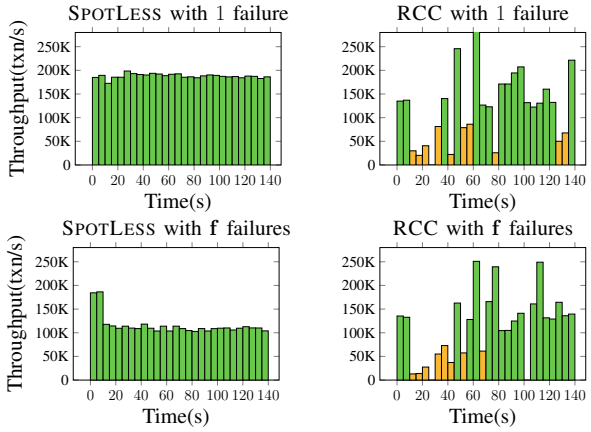

Fig. 12: Throughput timeline of SpotLess and RCC after injecting failures.

HotStuff decreases greatly. This is easily explained: the concurrent design of RCC and SpotLess load-balances the primary task to all replicas, whereas in Pbft and HotStuff the performance is bottlenecked by the bandwidth available to the single proposing primary.

As Figure 8 shows, non-responsive faulty replicas negatively affect the performance of SpotLess in all cases (Q4): indeed, non-responsive faulty replicas do not perform their primary role, due to which the non-faulty replicas can only wait until their timers expire to switch out these faulty primaries. The larger the number of replicas, the smaller the relative influence of faulty replicas on performance. For example, when there are $\mathbf{f}$ faulty replicas, the throughput of SpotLess128 decreases by 41% while that of SpotLess32 decreases by 54%. Due to *concurrent consensus*, a larger number of replicas implies more SpotLess instances with non-faulty primaries that utilize CPU resources while waiting for the instances with non-responsive primaries.

From Figure 7, 9, and 10, we also know that SpotLess shows great resilience to non-responsive faulty replicas when compared with other protocols (Q4). In a deployment with 128 replicas, the first 120 s after failures happen, SpotLess shows a gain in throughput over other protocols and a lower latency (Q1, Q2) despite the number of faulty replicas.

Thanks to the Ask-*recovery mechanism* and *Rapid View Synchronization*, described in Section III, SpotLess shows strong resilience to Byzantine attacks, as we can observe from the results in Figure 11. No matter the type of attack, the *victims* can quickly detect the failure and catch up by receiving $\mathbf{f}+1$ Sync messages from other non-faulty replicas and sending Ask messages. When facing non-responsive faulty replicas, the two mechanisms are useless, however, as timing out instances is the only way to advance view in this case.

The results of Figure 12 show obvious fluctuations in the real-time throughput of RCC after injecting failures. This is due to the usage of an exponential back-off penalty algorithm to ignore instances with faulty primaries. Eventually, the throughput of RCC gradually recovers to the original level and then keeps stable. This is the best case for RCC, however, as all $\mathbf{f}$ failures happen at the same time. If the failures appear one by one, RCC will suffer from these low-

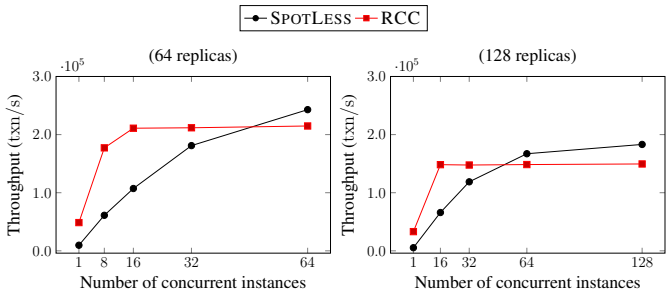

Fig. 13: Performance of SpotLess and RCC as a function of the number of concurrent instances.

throughput fluctuations (the yellow columns in Figure 12) during each failure. In contrast, SpotLess presents a more stable throughput timeline after failures happen (Q4).

Figure 13 shows that SpotLess benefits more from concurrent consensus than RCC, especially when there are many instances. (Q5). When there are 32 or fewer instances, RCC outperforms SpotLess because RCC enables *out-of-order processing* in individual instances, which is not supported by the chained design of SpotLess instances. As the number of concurrent instances increases, the throughput of RCC reaches a message processing bottleneck when there are 16 instances and then remains stable, whereas the throughput of SpotLess can increase further due to the lower message complexity and reaches its peak value when there are $\mathbf{n}$ instances, higher than RCC by up to 23%.

The performance of consensus is significantly influenced by computing and network resources. First, Figure 14(a) shows that the performance of all protocols decreases when compute power is restrictd (fewer CPU cores in each replica). Second, Figure 14(b) shows that decreasing the network bandwidth negatively impacts the performance of all protocols. We note that Narwhal-HS is barely affected, however, as it is limited by computing resources as it has to verify $\mathbf{n}-\mathbf{f}$ digital signatures per block. Similarly, Figure 14(c) and (d), show that *increasing* the number of regions, which not only decreases network bandwidth but also increases latency, *negatively impacts* the performance of all protocols. In all cases, SpotLess maintains a higher performance than RCC. Finally, the comparison between Figure 14(c) and (d), shows that that large batch sizes can partially mitigate bandwidth bottlenecks.

Figure 15 shows that the presence of Byzantine failures has similar negative effects on the performance of single-instance SpotLess and HotStuff. In all cases, the throughput of single-instance SpotLess is higher than that of HotStuff due to the lower computation costs of verifying signatures in SpotLess (as compared to dealing with the threshold signatures used in HotStuff). Hence, compared with HotStuff, replicas in SpotLess are able to respond more quickly, lowering the per-round latency and increasing throughput.

Based on our findings, we conclude that SpotLess makes full use of *concurrent consensus* (Q5), provides higher throughput and better scalability than any other consensus protocol (Q1), and does so with low latency (Q2). Moreover, client batching does benefit the performance of SpotLess (Q3). Finally, we conclude that SpotLess can efficiently deal with failures (Q4) thanks to *concurrent consensus*, the Ask-

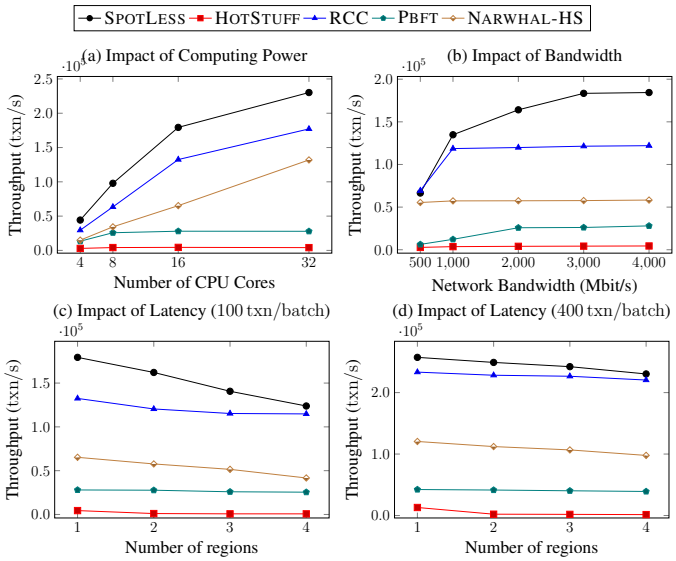

Fig. 14: Impact of computing and network resources on the performance with 128 replicas.

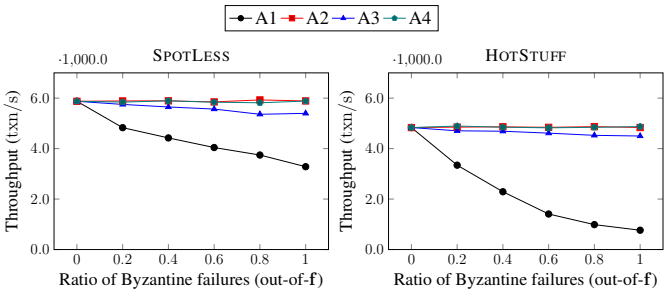

Fig. 15: Performance of single-instance SpotLess and HotStuff with failures.

recovery mechanism, and *Rapid View Synchronization*.

VI. Related Work

There is abundant literature on consensus and primary-backup consensus in specific (e.g., [18], [20], [26], [40]–[51]), to reduce the communication cost and improve performance and resilience of the consensus systems [35], [36], [52]–[59]. In previous sections, we already discussed how SpotLess relates to Pbft [3], RCC [23], HotStuff [24], and Narwhal-HS [31]. Next, we shall focus on other works that deal with either *improving throughput and scalability* or with *simplifying consensus*, the two strengths of SpotLess.

Leader-Less Consensus. Leader-less protocols such as HoneyBadger [60] and Dumbo [61] eliminate the limitations of Pbft and other primary-backup consensus protocols via a fully decentralized and fully asynchronous design. In these protocols, all replicas have the same responsibilities, due to which the cost of consensus is equally spread-out over all replicas. These leader-less protocols claim to improve resilience over Pbft in asynchronous environments. Due to the high complexity of fully asynchronous consensus, their practical performance is limited, however.

Sharding. Recently, there have been several approaches toward scalability of RDMSs by sharding them, e.g., [62]–[66]. Although sharding has the potential to drastically improve scalability for certain workloads, it does so at a high cost for complex workloads. Furthermore, sharding impacts resilience, as sharded systems put requirements on the number of failures in each individual shard (instead of putting those requirements on all replicas in the system). Finally, sharding is orthogonal to consensus, as proposed sharded systems all require a high-performance consensus protocol to run individual shards. For this task, SpotLess is an excellent candidate.

Reducing Primary Costs. There are several approaches toward reducing the cost for the primary to coordinate consensus in Pbft-style primary-backup consensus protocols, thereby reducing the limitations of primary-backup designs. Examples include (1) protocols such as FastBFT [67] and the geo-scale aware GeoBFT [68] that use a *hierarchical communication architecture* to offload the cost for a primary to propose client requests to other replicas; (2) protocols such as Narwhal-HS [31] that use a gossip-based communication protocol to replicate client requests, thereby sharply reducing the cost for primaries to propose these requests; and (3) protocols such as Algorand [69] that restrict consensus to a small subset of the replicas in the system (whom then enforce their decisions upon all other replicas), thereby reducing the cost of the primary to coordinate consensus. Unfortunately, each of these protocols introduces its own added complexity or environmental restrictions to the design of consensus, e.g., FastBFT requires trusted hardware, the design of GeoBFT impacts resilience in a similar way as sharded designs do, and Algorand relies on complex cryptographic primitives due to which it can only guarantee to work with high probability.

VII. Conclusion

In this paper, we proposed SpotLess, a high-performance robust consensus protocol. SpotLess combines the high throughput of *concurrent consensus architectures* with the reduced complexity provided by *chained consensus*. Furthermore, SpotLess improves on the resilience of existing chained consensus designs by introducing the *Rapid View Synchronization protocol*, which guarantees a continuous low-cost recovery path that is robust during unreliable communication and does not require costly threshold signatures.

We have put the design of SpotLess to the test by implementing it in Apache ResilientDB, our high-performance resilient fabric, and we compared SpotLess with existing consensus protocols. Our experiment results show that the performance of SpotLess is excellent: SpotLess greatly outperforms traditional primary-backup consensus protocols such as Pbft by up to 430%, Narwhal-HS by up to 137%, and HotStuff by up to 3803%. SpotLess is even able to outperform RCC, a state-of-the-art high-throughput concurrent consensus protocol, by up to 23% in optimal conditions, while providing lower latency in all cases. Furthermore, SpotLess can maintain a stable latency and consistently high throughput, even during failures.

Acknowledgement

This work is partially funded by NSF Award Numbers 2245373 and 2112345.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] A. Narayanan and J. Clark, "Bitcoin's academic pedigree," *Communications of the ACM*, vol. 60, no. 12, pp. 36–45, 2017.

[3] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[4] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[5] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: the works of leslie lamport*, 2019, pp. 203–226.

[6] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[7] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[8] M. N. Kamel Boulos, J. T. Wilson, and K. A. Clauson, "Geospatial blockchain: promises, challenges, and scenarios in health and healthcare," *International Journal of Health Geographics*, vol. 17, no. 1, pp. 1211–1220, 2018.

[9] L. Lao, Z. Li, S. Hou, B. Xiao, S. Guo, and Y. Yang, "A survey of iot applications in blockchain systems: Architecture, consensus, and traffic modeling," *ACM Comput. Surv.*, vol. 53, no. 1, 2020.

[10] A. Rejeb, J. G. Keogh, S. Zailani, H. Treiblmaier, and K. Rejeb, "Blockchain technology in the food industry: A review of potentials, challenges and future research directions," *Logistics*, vol. 4, no. 4, 2020.

[11] H. Treiblmaier and R. Beck, Eds., *Business Transformation through Blockchain*. Springer, 2019.

[12] P. Ruan, T. T. A. Dinh, D. Loghin, M. Zhang, and G. Chen, *Blockchains: Decentralized and Verifiable Data Systems*. Springer, 2022.

[13] M. Herlihy, "Blockchains from a distributed computing perspective," *Commun. ACM*, vol. 62, no. 2, pp. 78–85, 2019.

[14] G. Wood, "Ethereum: a secure decentralised generalised transaction ledger," 2016, EIP-150 revision. [Online]. Available: https://gavwood.com/paper.pdf

[15] A. de Vries, "Bitcoin's growing energy problem," *Joule*, vol. 2, no. 5, pp. 801–805, 2018.

[16] H. Vranken, "Sustainability of bitcoin and blockchains," *Current Opinion in Environmental Sustainability*, vol. 28, pp. 1–9, 2017.

[17] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," *Commun. ACM*, vol. 61, no. 7, pp. 95–102, 2018.

[18] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "SBFT: A scalable and decentralized trust infrastructure," in *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 568–580.

[19] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 2019, pp. 347–356.

[20] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, pp. 7:1–7:39, 2009.

[21] F. Nawab and M. Sadoghi, 2023.

[22] S. Gupta, S. Rahnama, S. Pandey, N. Crooks, and M. Sadoghi, "Dissecting BFT consensus: In trusted components we trust!" 2022. [Online]. Available: https://arxiv.org/abs/2202.01354

[23] S. Gupta, J. Hellings, and M. Sadoghi, "RCC: resilient concurrent consensus for high-throughput secure transaction processing," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 1392–1403.

[24] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. ACM, 2019, pp. 347–356.

[25] S. Gupta, J. Hellings, and M. Sadoghi, *Fault-Tolerant Distributed Transactions on Blockchain*, ser. Synthesis Lectures on Data Management. Morgan & Claypool, 2021.

[26] C. Stathakopoulou, T. David, and M. Vukolic, "Mir-BFT: High-throughput BFT for blockchains," 2019. [Online]. Available: http://arxiv.org/abs/1906.05552

[27] C. Stathakopoulou, M. Pavlovic, and M. Vukolić, "State machine replication scalability made simple," in *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, 2022, pp. 17–33.

[28] Y. Baek, Joonsang Zheng, "Simple and efficient threshold cryptosystem from the gap diffie-hellman group," in *GLOBECOM '03. IEEE Global Telecommunications Conference*, vol. 3. IEEE, 2003, pp. 1491–1495.

[29] I. Abraham, N. Crooks, N. Giridharan, H. Howard, and F. Suri-Payer, "It's not easy to relax: liveness in chained BFT protocols," 2022. [Online]. Available: https://arxiv.org/abs/2205.11652

[30] G. Zhang, F. Pan, S. Tijanic, and H.-A. Jacobsen, "Prestigebft: Revolutionizing view changes in bft consensus algorithms with reputation mechanisms," 2023.

[31] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus," in *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, 2022, pp. 34–50.

[32] G. Tel, *Introduction to Distributed Algorithms*, 2nd ed. Cambridge University Press, 2001.

[33] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 2nd ed. Chapman and Hall/CRC, 2014.

[34] D. Kang, S. Rahnama, J. Hellings, and M. Sadoghi, "SpotLess: Concurrent rotational consensus made practical through rapid view synchronization," 2023. [Online]. Available: https://arxiv.org/abs/2302.02118

[35] P. Civit, M. A. Dzulfikar, S. Gilbert, V. Gramoli, R. Guerraoui, J. Komatovic, and M. Vidigueira, "Byzantine consensus is $\theta(\mathbf{n}^2)$: The dolev-reischuk bound is tight even in partial synchrony!" in *36th International Symposium on Distributed Computing (DISC 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 246. Schloss Dagstuhl, 2022, pp. 14:1–14:21.

[36] A. Lewis-Pye, "Quadratic worst-case message complexity for state machine replication in the partial synchrony model," 2022. [Online]. Available: https://arxiv.org/abs/2201.01107

[37] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 2010, pp. 143–154.

[38] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "BLOCKBENCH: A framework for analyzing private blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1085–1100.

[39] C. Tsaousis and P. Whineray, "FireHOL–Linux firewalling and traffic shaping for humans," 2023. [Online]. Available: https://firehol.org/

[40] C. Cachin and M. Vukolic, "Blockchain consensus protocols in the wild (keynote talk)," in *31st International Symposium on Distributed Computing*, vol. 91. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 1:1–1:16.

[41] C. Berger and H. P. Reiser, "Scaling byzantine consensus: A broad analysis," in *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. ACM, 2018, pp. 13–18.

[42] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang, "Untangling blockchain: A data processing view of blockchain systems," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 7, pp. 1366–1385, 2018.

[43] K. Antoniadis, A. Desjardins, V. Gramoli, R. Guerraoui, and I. Zablotchi, "Leaderless consensus," in *41st IEEE International Conference on Distributed Computing Systems*. IEEE, 2021, pp. 392–402.

[44] P. Aublin, R. Guerraoui, N. Knezevic, V. Quéma, and M. Vukolic, "The next 700 BFT protocols," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, pp. 12:1–12:45, 2015.

[45] C. A. Ardagna, M. Anisetti, B. Carminati, E. Damiani, E. Ferrari, and C. Rondanini, "A blockchain-based trustworthy certification process for composite services," in *2020 IEEE International Conference on Services Computing (SCC)*. IEEE, 2020, pp. 422–429.

[46] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic, "XFT: Practical fault tolerance beyond crashes," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. USA: USENIX Association, 2016, pp. 485–500.

[47] D. Loghin, T. T. A. Dinh, A. Maw, C. Gang, Y. M. Teo, and B. C. Ooi, "Blockchain goes green? part ii: Characterizing the performance and cost of blockchains on the cloud and at the edge," 2022. [Online]. Available: https://arxiv.org/abs/2205.06941

[48] C. Rondanini, B. Carminati, F. Daidone, and E. Ferrari, "Blockchain-based controlled information sharing in inter-organizational workflows,"

in *2020 IEEE International Conference on Services Computing (SCC)*. IEEE, 2020, pp. 378–385.

[49] P. Ruan, T. T. A. Dinh, Q. Lin, M. Zhang, G. Chen, and B. C. Ooi, "Lineagechain: a fine-grained, secure and efficient data provenance system for blockchains," *VLDB J.*, vol. 30, no. 1, pp. 3–24, 2021.

[50] P. Sheng, G. Wang, K. Nayak, S. Kannan, and P. Viswanath, "BFT protocol forensics," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2021, pp. 1722–1743.

[51] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi, "Gem^2-tree: A gas-efficient structure for authenticated range queries in blockchain," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 842–853.

[52] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *Proceedings of the 25th USENIX Conference on Security Symposium*. USENIX, 2016, pp. 279–296.

[53] E. Blass and F. Kerschbaum, "BOREALIS: building block for sealed bid auctions on blockchains," in *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security*. ACM, 2020, pp. 558–571.

[54] R. Yuan, Y. Xia, H. Chen, B. Zang, and J. Xie, "ShadowEth: Private smart contract on public blockchain," *J. Comput. Sci. Technol.*, vol. 33, no. 3, pp. 542–556, 2018.

[55] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. ACM, 2020, pp. 955–970.

[56] V. A. Sartakov, S. Brenner, S. B. Mokhtar, S. Bouchenak, G. Thomas, and R. Kapitza, "Eactors: Fast and flexible trusted computing using SGX," in *Proceedings of the 19th International Middleware Conference*, P. Ferreira and L. Shrira, Eds. ACM, 2018, pp. 187–200.

[57] M. Sit, M. Bravo, and Z. István, "An experimental framework for improving the performance of BFT consensus for future permissioned blockchains," in *DEBS '21: The 15th ACM International Conference on Distributed and Event-based Systems, Virtual Event, Italy, June 28 - July 2, 2021*. ACM, 2021, pp. 55–65.

[58] M. F. Madsen, M. Gaub, M. E. Kirkbro, and S. Debois, "Transforming byzantine faults using a trusted execution environment," in *15th European Dependable Computing Conference*. IEEE, 2019, pp. 63–70.

[59] L. Kuhring, Z. István, A. Sorniotti, and M. Vukolić, "StreamChain: Building a low-latency permissioned blockchain for enterprise use-cases," in *2021 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2021, pp. 130–139.

[60] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 31–42.

[61] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous BFT protocols," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2020, pp. 803–818.

[62] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *Proceedings of the 2019 International Conference on Management of Data*. ACM, 2019, pp. 123–140.

[63] J. Hellings and M. Sadoghi, "ByShard: sharding in a byzantine environment," *The VLDB Journal*, vol. 32, no. 6, pp. 1343–1367, 2023.

[64] ——, "Byshard: Sharding in a byzantine environment," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2230–2243, 2021.

[65] M. J. Amiri, D. Agrawal, and A. El Abbadi, "SharPer: Sharding permissioned blockchains over network clusters," 2019. [Online]. Available: https://arxiv.org/abs/1910.00765v1

[66] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy, "BlockchainDB: A shared database on blockchains," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1597–1609, 2019.

[67] J. Liu, W. Li, G. O. Karame, and N. Asokan, "Scalable byzantine consensus via hardware-assisted secret sharing," *IEEE Trans. Comput.*, vol. 68, no. 1, pp. 139–151, 2019.

[68] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, "ResilientDB: Global scale resilient blockchain fabric," *Proc. VLDB Endow.*, vol. 13, no. 6, pp. 868–883, 2020.

[69] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP. ACM, 2017, pp. 51–68.