Taming the Contention in Consensus-based Distributed Systems

Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran

Abstract—Contention plays a crucial role in the design of consensus protocols. State-of-the-art solutions optimize their performance for either very low or high contention situations. We propose CAESAR, a novel multi-leader Generalized Consensus protocol, most suitable for geographical replication, that is optimized for low-to-moderate contention. With an evaluation study, we show that CAESAR outperforms other multi-leader (e.g., EPaxos) and single-leader (e.g., Multi-Paxos) competitors by up to 1.7x and 3.5x, respectively, in the presence of 30% conflicting requests, in a geo-replicated setting. Furthermore, we acknowledge that there is no one-size-fits- all consensus solution, especially for all levels of contentious workloads. Thus, we also propose SPECTRUM, a consensus framework that is able to switch consensus protocols at runtime to enable a dynamic reaction to changes in the workload and deployment characteristics. We show empirically that SPECTRUM can guarantee high availability even during periods of transition between consensus protocols.

Index Terms—Distributed systems, fault tolerance, consensus, leaderless consensus, contention-agnostic consensus

1 INTRODUCTION

GEOGRAPHICALLY replicated services, namely those where actors are spread across geographic locations and operate on the same shared database, can be implemented in an easier manner by exploiting underlying synchronization mechanisms that provide strong consistency guarantees. These mechanisms ultimately rely on implementations of Consensus [1] to globally agree on sequences of operations to be executed. Paxos [2], [3] is a very popular algorithm for solving Consensus among participants interconnected by asynchronous networks, even in presence of faults, and it can be leveraged for building such robust and strongly consistent services easily [4], [5], [6], [7], [8]. An example of Paxos used in a production system is Google Spanner [4].

In the most deployed version of Paxos, Multi-Paxos [3], a designated node is elected as the leader and is responsible for deciding the order of client-issued commands. Multi-Paxos solves consensus in only three communication delays, however, in practice, its performance is tied to the performance of the leader, and this relation is particularly risky when Mulit-Paxos is deployed in geo-scale because network delays can be arbitrarily large and unpredictable [9]. In these settings, the leader might be unreachable or often slow, thus causing the slow down of the entire system.

- *B. Arun and B. Ravindran are with the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, 24061. E-mail: {balajia, binoy}@vt.edu.*
- *S. Peluso is with Facebook Inc., Seattle, WA 98109. Email: sebapeluso@fb.edu.*
- *R. Palmieri is with Lehigh University, Bethlehem, PA 18015. Email: palmieri@lehigh.edu.*
- *G. Losa is with Galois Inc., Portland, OR 97203. Email: giuliano@losa.fr. The work was done while the authors were at Virginia Tech.*

Manuscript received 18 Feb. 2019; revised 25 Dec. 2019; accepted 12 Jan. 2020.

To overcome this limitation, protocols aimed at allowing multiple nodes to operate as command leaders simultaneously [10], [11], [12] have been proposed. Such solutions provide implementations of Generalized Consensus [13], a variant of consensus generalized to agree on a common order of non-commuting (or conflicting[1]) commands. These approaches, despite avoiding the single-leader bottleneck, suffer from other types of costs whenever a non-trivial amount of conflicting commands (e.g., 5% – 40%) is proposed concurrently as they do not rely on a unique point of decision.

CAESAR. This paper presents the first multi-leader implementation of Generalized Consensus designed for maintaining high performance in the presence of both mostly non-conflicting workloads (named as such if less than 5% of conflicting commands are issued) and conflicting workloads (where at most 40% of commands conflict with each other). For this reason, our solution is apt for geo-scale deployments. More specifically, State-of-the-art implementations of Generalized Consensus (e.g., EPaxos [10] and M^2Paxos [14]) reduce the minimum number of communication delays required to reach an agreement from three to two in case a proposed command does not encounter any contention (*fast decision*). However, they fail in the following two aspects: they are not able to minimize the latency as soon as some contention on issued commands arises, with the consequence of requiring a *slow decision*, which consists of at least four communication delays; or they adopt complex conflict resolution mechanisms to find the final order of commands, which results in a noticeable bottleneck before delivery.

To address these aspects, we propose CAESAR, a consensus protocol that deploys an innovative multi-leader ordering scheme. As a high-level intuition, when a conflicting command c is proposed, CAESAR seeks an agreement on a common delivery timestamp rather than on its set of conflicting commands, unlike existing solutions. To facilitate this, a local *wait condition* is deployed to prevent commands

1. Contention is defined as the amount of conflicting commands

conflicting with c from interfering with the decision process of c if they have a timestamp greater than c's.

The core idea behind the ordering process of CAESAR is the following: a command is associated with a logical timestamp by the sender, and when a quorum of nodes confirms the timestamp's validity, the command is ordered after all the conflicting commands having a valid earlier timestamp. Otherwise, the timestamp is considered invalid and the command is rejected forcing it to undergo two more communication delays (total of four) before being decided. With this scheme, CAESAR boosts timestamp-based ordering protocols, such as Mencius, by enabling the exploitation of quorums, a fundamental requirement in geo-scale where contacting all nodes is not feasible. Moreover, CAESAR does not rely on a single designated leader, unlike Multi-Paxos.

Our approach also provides the benefit of a much simpler and more parallel delivery of ordered commands compared to EPaxos [10], a state-of-the-art approach that requires an analysis of the dependency graphs. That is because, once the delivery timestamp for a command is finalized, the command implicitly carries with itself the set of predecessor commands that have to be delivered before it. This so called *predecessors set* is computed during the execution of the ordering algorithm for the decision of the timestamp, and not after the delivery of the command; therefore, no extra local computation is needed.

We implemented CAESAR in Java and conducted an evaluation study using key-value store benchmark. We injected different workloads by varying the percentage of conflicting commands and measure various performance parameters. As a testbed, we used the AWS EC2 infrastructure. We contrasted CAESAR against: EPaxos [10] and M^2Paxos [14], multi-leader quorum-based Generalized Consensus implementations; Mencius [11], a multi-leader timestamp-based protocol that does not rely on quorums in the fast path; Multi-Paxos [2], a single-leader protocol. The results confirm the effectiveness of CAESAR in providing *fast decisions*, even in the presence of conflicting workloads, while competitors slow down. Using workloads with a conflict percentage in the range of 2% – 50%, CAESAR outperforms EPaxos, which is the closest competitor in most of the cases, by reducing latency as much as 60% and increasing throughput by 1.7×.

SPECTRUM. The evaluation of CAESAR and it's competitors (see § 3.5) concluded that the efficiency of leaderless protocols depends on the contention level. In short, M^2Paxos and EPaxos are better suited in low contention, while CAESAR is better under low-to-medium contention. Furthermore, it was observed that these protocols could be outperformed by single leader-based ones, e.g., Multi-Paxos, when the contention is really high, since the complex mechanisms that are adopted to establish the order of non-commuting commands can result in a remarkable and expensive overhead [12], [14], which a performance improvement cannot amortize even in configurations where commands mostly commute.

We acknowledge that there is no one-size-fits-all consensus protocol to address different workload and deployment scenarios. CAESAR is no exception. Therefore, we propose SPECTRUM, a novel, general consensus framework that provides a protocol-agnostic switching scheme that is powered by an oracle to react to changing contention levels and network latencies, by using the right consensus protocol for that scenario. SPECTRUM can switch protocols online, without any downtime, in a way completely oblivious to the clients, and while tolerating faults.

We implemented and evaluated SPECTRUM by onboarding the Multi-Paxos, CAESAR [15] and M^2PAXOS protocols. By injecting different conflicting workloads at runtime, we show that SPECTRUM adds no overhead to the underlying consensus protocols during the time period when the workload is stable; moreover, it is able to limit the increase of user perceived latency when switching among consensus protocols, compared to an off-the-shelf *stop-and-restart* solution. SPECTRUM does not reject any command during the switch, unlike the *stop-and-restart* solution which causes a downtime of 3–4 seconds.

The main contributions of this work are:

- CAESAR, the first Generalized Consensus protocol to provide fast decisions under moderate contention.
- SPECTRUM, a framework for transitioning consensus protocols at runtime, while maintaining high availability.

The rest of this paper is organized as follows. Section 2 presents the system model. Section 3 overviews, details and evaluates CAESAR. Section 4 motivates, describes, and evaluates SPECTRUM and its switching mechanism. Section 5 describes related work and Section 6 concludes the paper.

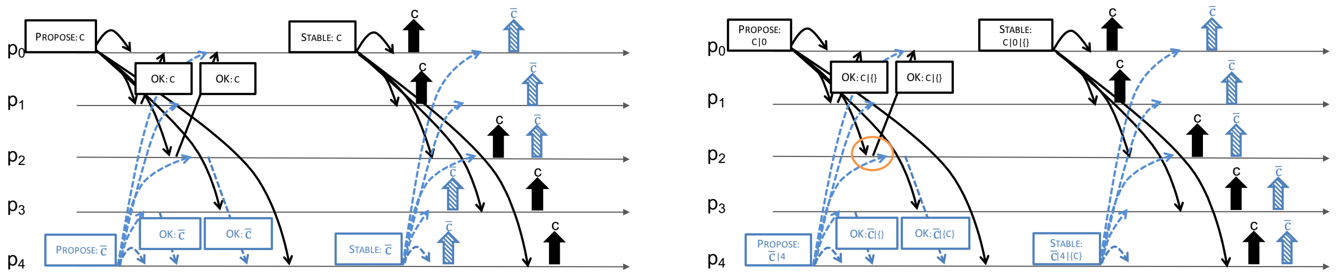
2 SYSTEM MODEL

We assume a set of nodes $\Pi = \{p_1, p_2, \ldots, p_N\}$ that communicate through message passing and do not have access to either a shared memory or a global clock. Nodes may fail by crashing but do not behave maliciously. A node that does not crash is called correct; otherwise, it is faulty. Messages may experience arbitrarily long (but finite) delays.

To circumvent the FLP [16] result, we assume that the system can be enhanced with the weakest type of unreliable failure detector [17] that is necessary to implement a leader election service [18]. In addition, we assume that at least a strict majority of nodes, i.e., $\lfloor \frac{N}{2} \rfloor + 1$, is correct. We name *classic quorum* ($\mathcal{CQ}$), or more simply *quorum*, any subset of Π with size at least equal to $\lfloor \frac{N}{2} \rfloor + 1$. We name *fast quorum* ($\mathcal{FQ}$) any subset of Π with size at least equal to $\lceil \frac{3N}{4} \rceil$ (derived by minimizing $\mathcal{CQ}$). Some protocols like CAESAR require a fast quorum to achieve fast decisions in two communication delays, while a classic quorum is required whenever the protocol needs more than two communication delays to reach a decision.

We follow the definition of Generalized Consensus [13]: each node can propose a command c via the PROPOSE(c) interface, and nodes decide command structures *C-struct* cs via the DECIDE(cs) interface. The specification is such that: commands that are included in decided *C-structs* must have been proposed (*Non-triviality*); if a node decided a *C-struct* v at any time, then at all later times it can only decide $v \bullet \sigma$, where σ is a sequence of commands (*Stability*); if c has been proposed then c will be eventually decided in some *C-struct* (*Liveness*); and two *C-structs* decided by two different nodes are prefixes of the same *C-struct* (*Consistency*). Note that the symbol $\bullet$ is the append operator as defined in [13].

For simplicity of the presentation, we also use the notation DECIDE(c) for the decision of a command c at any node, with the following semantics: the sequence of k consecutive



(a) The non-commutative commands c and $\bar{c}$ are executed only after a quorum of nodes receives them. A total order of the commands is not enforced in this case, since commands are submitted via fault-tolerant broadcast.

(b) The non-commutative commands c and $\bar{c}$ are executed only after a quorum of nodes receives them. A total order of the commands is enforced in this case: $\bar{c}$ is executed after c on all nodes, since $\mathcal{T} = 0 < \bar{\mathcal{T}} = 4$, and timestamp are received in order by p_2.

Fig. 1. Fault-tolerant broadcast execution vs. CAESAR execution

calls of DECIDE$(c_1) \bullet$ DECIDE$(c_2) \bullet \cdots \bullet$ DECIDE(c_k) is equivalent to the call of DECIDE$(c_1 \bullet c_2 \bullet \cdots \bullet c_k)$.

We say that two commands c and $\bar{c}$ are *non-commutative*, or *conflicting*, and we write $c \sim \bar{c}$, if the results of the execution of both c and $\bar{c}$ depend on whether c has been executed before or after $\bar{c}$. It is worth noting that, as specified in [13], two *C-structs* are equal if they order conflicting commands in the same way.

3 CAESAR

In this section, we present CAESAR, a protocol to solve consensus efficiently in low-to-moderate contention scenarios. We begin by providing an overview of the protocol, followed by a detailed algorithm description and evaluation.

3.1 Overview

We introduce CAESAR incrementally by starting from a base protocol, which only provides reliable broadcast of commands, and then we present the design of the final protocol, which implements Generalized Consensus. We consider the first protocol as a reference point to show the minimal costs that are required to implement our specification of Consensus, and we explain how CAESAR is able to maximize the probability to execute with the same number of communication steps as the reference protocol. Section 3.4 provides the details of CAESAR.

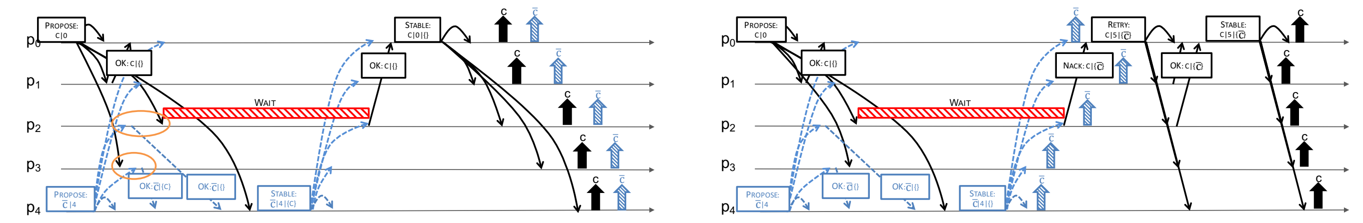
A necessary condition for implementing both a reliable broadcast protocol and the *consistency* property of CAESAR is guaranteeing that if a command is delivered to a (correct or faulty) node, then it is eventually delivered to any correct node. This is because whenever a command is executed by a node and the result externalized to clients, the command must be durable in the system despite crashes.

The base protocol executes as shown in Figure 1(a). When a client proposes a command c to the system via the interface PROPOSE(c), the protocol chooses a node to be c's leader, p_0 in this case, which broadcasts a PROPOSE message with c to all nodes. Afterwards, whenever c's leader collects a quorum of $\mathcal{OK}$ replies for c, it broadcasts a STABLE message for c in order to allow all nodes (including the leader itself) to deliver and execute c (thick arrows in Figure 1(a)).

The base protocol is fault-tolerant because whenever c is delivered and executed on some node, one of the following conditions is true, regardless of the crash of f nodes: if c's leader does not crash, eventually any other correct node receives the STABLE message for c; or if c's leader crashes, there always exists at least one correct node that received the PROPOSE message for c, so it can take over the crashed leader by re-executing the protocol for c. Moreover, the scheme adopted by the base protocol needs two communication delays: one for the PROPOSE message and one for the $\mathcal{OK}$ messages, to return the result of an execution to a client. Two communication delays are the minimum required to implement consensus in an asynchronous system [19].

The base protocol does not implement Generalized Consensus because it does not enforce any order on the delivery of non-commutative commands. In fact, two concurrent commands, c and $\bar{c}$, can be delivered and executed in any order by different nodes, regardless of their commutativity relation. CAESAR implements the specification of Generalized Consensus by building a novel timestamp-based mechanism on top of the base protocol to enforce a total order among non-commutative commands. We still rely on Figure 1 for showing the intuition. Command c is associated with a unique logical *timestamp* $\mathcal{T}$ (see Section 3.4.1 for the timestamp assignment), and it can be delivered and executed only after a quorum of nodes confirms that no other command $\bar{c}$ with timestamp $\bar{\mathcal{T}}$, where $\bar{c} \sim c$ and $\bar{\mathcal{T}} > \mathcal{T}$, will be executed before c. Note that in this section we do not distinguish between fast and classic quorums, although in Section 3.4 we explain that a fast quorum is required at this stage due to the lower-bound defined in [19]. Here, we assume c's leader does not fail or is suspected; the case of faulty leaders is discussed in Section 3.4.5.

Figure 1(b) shows how CAESAR applies this idea to the execution of Figure 1(a). Node p_0 broadcasts c by proposing it with timestamp 0; then a quorum of nodes confirms c since none of those nodes has already received $\bar{c}$ with a timestamp greater than 0. The confirmation from a process p_j is sent via an $\mathcal{OK}$ message, which, unlike the base protocol, includes a *predecessors set* $\mathcal{P}red_j$ of the commands observed by p_j, and that should precede c. When p_4 broadcasts $\bar{c}$ with timestamp 4, it receives a quorum of replies from p_2, p_3, p_4, which confirms that $\bar{c}$ can be executed with timestamp 4 and only after c has been executed. This happens because p_2 already observed c at the time it received $\bar{c}$ (see circle in Figure 1(b)), and it included $\mathcal{P}red_2 = \{c\}$ in the $\mathcal{OK}$



(a) p_2 sends an $\mathcal{OK}$ message for c at timestamp $\mathcal{T} = 0$ because c is in the predecessors set of $\bar{c}$, and $\bar{c}$ is decided at timestamp $\bar{\mathcal{T}} = 4$.

(b) p_2 rejects c at timestamp $\mathcal{T} = 0$ because c is not in the predecessors set of $\bar{c}$, and $\bar{c}$ is decided at timestamp $\bar{\mathcal{T}} = 4$. c is decided at timestamp 5 after a retry.

Fig. 2. Execution of the wait condition in CAESAR due to out of order reception of non-commutative commands on node p_2. Command c waits for command $\bar{c}$ to be stable on node p_2, since c's timestamp $\mathcal{T}$ has been received after $\bar{c}$'s timestamp $\bar{\mathcal{T}}$, and $\mathcal{T} = 0 < \bar{\mathcal{T}} = 4$.

message for $\bar{c}$. A command leader can broadcast the STABLE message as soon as it receives a quorum of $\mathcal{OK}$ messages for that command, and it also includes the timestamp and the set $\mathcal{P}red$, which is the union of the predecessors sets received in the $\mathcal{OK}$ messages. Therefore, in CAESAR, unlike the base protocol, a node can execute c when it receives the STABLE message for c and only after it has executed all the commands in c's $\mathcal{P}red$.

As shown in Figure 1(b), a command's leader in CAESAR guarantees a *fast decision* in two communication delays as long as the proposed timestamp is confirmed by a quorum of nodes and despite the non-uniform replies that it collected (the set of predecessors collected by p_4 for $\bar{c}$ is different). This also constitutes a significant difference between CAESAR and other state-of-the-art Generalized Consensus implementations, e.g., EPaxos, which require at least two additional communication delays before the execution of $\bar{c}$ in the example of Figure 1(b).

In the following, we answer two questions: what does a node do if it observes out of order timestamps (Section 3.2)? How does a command's leader behave if a node in the replying quorum rejects a proposed timestamp (Section 3.3)?

3.2 Out of Order Timestamps

Let us now consider the scenario in Figure 2(a), where, unlike the one in Figure 1, node p_2 receives the PROPOSE for c after having received the one for $\bar{c}$ (see the circle on p_2). In this case, p_2 cannot directly send an $\mathcal{OK}$ message for c, because $\mathcal{T} = 0 < \bar{\mathcal{T}} = 4$, and $\bar{c}$ could be finally decided at timestamp $\bar{\mathcal{T}}$ without ever considering c as its predecessor, and hence be executed before c, with a resulting violation of the order of the timestamps. On the other hand, sending a rejection for c would require additional communication delays, because c's leader would be forced to retry the decision procedure with a new timestamp. This overhead is unnecessary if c was received before $\bar{c}$ on another node, which could be part of the quorum of replies to $\bar{c}$'s leader.

In this case, CAESAR enforces a *wait condition* for c on p_2 (bar labelled *wait* along p_2's timeline in Figure 2(a)) in order to prevent the execution of any step for c until p_2 receives the final decision for $\bar{c}$. Afterwards, if the final decision for $\bar{c}$ includes c in $\bar{c}$'s $\mathcal{P}red$, p_2 can reply with an $\mathcal{OK}$ message to c's leader. As a result, CAESAR is able to increase the probability of deciding commands in two communication delays even in the case of out of order reception of timestamps. Note that the *wait condition* does not cause deadlock since only commands with a lower timestamp, e.g., c, wait for the final decision of conflicting commands with a higher timestamp, e.g., $\bar{c}$.

3.3 Rejection of Timestamps

In case a node cannot confirm a timestamp $\mathcal{T}$ proposed for a command c, it sends a rejection $\mathcal{NACK}$ to c's leader, forcing the leader to retry c with a timestamp greater than $\mathcal{T}$. This is the case of Figure 2(b), where p_2 rejects $\mathcal{T} = 0$ for c because it already received the STABLE message for $\bar{c}$ with timestamp $\bar{\mathcal{T}} > \mathcal{T}$ and c is not in $\bar{c}$'s $\mathcal{P}red$. p_2 also sends back the set of commands that caused the rejection (i.e., $\bar{c}$) to aid in choosing the next timestamp for c.

In CAESAR, if a command's leader receives at least one $\mathcal{NACK}$ message for the proposed command c, it assigns a new timestamp $\mathcal{T}_{new}$ greater than any suggestion received in the $\mathcal{NACK}$ messages, and it broadcasts a RETRY message to ask for the acceptance of $\mathcal{T}_{new}$ to a quorum of nodes. Note that if a node sends a $\mathcal{NACK}$ message for a command c to c's leader, it means that c's leader would receive at least a $\mathcal{NACK}$ message for c from any other quorum due to the way a command rejection is computed (see Section 3.4).

The RETRY message also contains the predecessors set $\mathcal{P}red$, which is computed as the union of predecessors received in the quorum of replies from the previous phase, as the case of Section 3.2. Therefore, in Figure 2(b), p_0 broadcasts the RETRY with timestamp $\mathcal{T}_{new} = 5$ and $\mathcal{P}red = \{\bar{c}\}$ for c.

Retrying a command with a new timestamp does not entail restarting the procedure from the beginning. In fact, unlike the case of a PROPOSE message, CAESAR guarantees that a RETRY message can never be rejected (see Sections 3.4.3 and 4.3). Such a guarantee ensures starvation-free agreement of commands. The reply to a RETRY message for c could contain a set of additional predecessors that were not received by c's leader during the previous communication phase. This set is sent along with the STABLE message for c.

3.4 Protocol Details

A command c that is proposed to CAESAR can go through *four phases* before it gets decided and the outcome of its execution is returned to the client. CAESAR schedules the execution of those four phases in order to provide *two modes* of decision, called *fast decision* and *slow decision*.

A command c is proposed by one of the nodes, which assumes the role of c's *leader* and coordinates the decision of c by starting the *fast proposal phase*. If this phase returns a positive outcome after having collected replies from a quorum of $\mathcal{FQ}$ nodes, the leader can execute the final *stable phase*, which finalizes the decision of c as a *fast decision*, with a latency of two communication delays. Otherwise, if the *fast proposal phase* returns a negative outcome, the leader executes an additional *retry phase*, in which it contacts a quorum of $\mathcal{CQ}$ nodes, before issuing the final *stable phase*. This results in a *slow decision*, with a latency of four communication delays.

In this section we describe CAESAR by detailing the required data structures in Section 3.4.1, the procedure for a fast decision in Section 3.4.2, the procedure for a slow decision in Section 3.4.3, and the behavior of the protocol in case of failures in Section 3.4.5. We also explain how CAESAR behaves in case a leader is not able to contact a fast quorum of nodes during the execution of the *fast proposal phase* for a command, as long as no more than f nodes crash. This case entails the execution of an additional *slow proposal phase* after the *fast proposal phase* and before the remaining *retry* and *stable phases*. This part is overviewed in Section 3.4.4 and detailed in the technical report [20].

In Figure 4 we provide the main pseudocode of CAESAR for the decision of a command c. Each horizontal block of the figure is a phase, and phases are linked through arrows to indicate the transition from one phase to another. For instance, in case of fast decision, we have a transition from the fast proposal phase to the stable phase; on the other hand all the other transitions are part of a slow decision. Moreover, the pseudocode is vertically partitioned in order to distinguish the part that is executed by the command c's leader and the part that can be executed by any node (including the leader); it is also named as acceptor following the Generalized Consensus framework. Finally, the pseudocodes of auxiliary functions and the recovery from a failure are provided in Figures 3 and 5, respectively.

3.4.1 *Data Structures per node p_i*

$\underline{\mathcal{TS}_i}$. It is a logical clock with monotonically increasing values in a totally ordered set of elements, and it is used to generate timestamps for the commands that are proposed by p_i. Its value at a certain time is greater than the timestamp of any command that has been handled by p_i before that time.

We assume that whenever p_i sends a command, $\mathcal{TS}_i$ is updated with a greater value and used as timestamp $\mathcal{T}$ for the command. Also, whenever p_i receives a command with timestamp $\mathcal{T}$, it updates its $\mathcal{TS}_i$ with a value that is greater than $\mathcal{T}$, if $\mathcal{T} \geq \mathcal{TS}_i$. We also assume that for any two $\mathcal{TS}_i$ and $\mathcal{TS}_j$, of p_i and p_j respectively, the value of $\mathcal{TS}_i$ is different from the value of $\mathcal{TS}_j$ at any time. This is guaranteed by choosing the values of $\mathcal{TS}_i$ ($\mathcal{TS}_j$, respectively) in the set $\{\langle k, i\rangle : k \in \mathbb{N}\}$ ($\{\langle k, j\rangle : k \in \mathbb{N}\}$, respectively). The total order relation on those values is defined as follows: for any two $\langle k_1, i\rangle$, $\langle k_2, j\rangle$, we have that $\langle k_1, i\rangle < \langle k_2, j\rangle \Leftrightarrow k_1 < k_2 \vee (k_1 = k_2 \wedge i < j)$. The initial value of $\mathcal{TS}_i$ is $\langle 0, i\rangle$.

$\underline{\mathcal{H}_i}$. It is the data structure recording the status of commands seen by p_i. It is represented as a map of tuples of the form $\langle c, \mathcal{T}, \mathcal{P}red, status, \mathcal{B}, forced\rangle$ where: c is a command; $\mathcal{T}$ is the latest timestamp of c; $\mathcal{P}red$ is the set of commands that should precede c in the final decision; $status$ is the current status of c, and it has values in the set $\{fast\text{-}pending, slow\text{-}pending, accepted, rejected, stable\}$; $\mathcal{B}$ is the ballot number associated with this event, and it has values in $\mathbb{N}$; and $forced$ is a boolean variable with values in $\{\top, \bot\}$, and it indicates if the info associated with this event (e.g., $\mathcal{P}red$) has been forced by a recovery procedure.

Each tuple in $\mathcal{H}_i$ is uniquely identified by the first element of the tuple, i.e., the command, and thus $\mathcal{H}_i$ contains at most one tuple per command. For a more compact representation, we use "$-$" as the wildcard symbol whenever we are not interested in the value of a specific element of a tuple.

We also use the following notations: $\mathcal{H}_i$.UPDATE(c, $\mathcal{T}$, $\mathcal{P}red$, $status$, $\mathcal{B}$, $forced$) to indicate that the protocol appends the tuple $\langle c, \mathcal{T}, \mathcal{P}red, status, \mathcal{B}, forced\rangle$ to $\mathcal{H}_i$, by first possibly deleting any existing tuple $\langle c, -, -, -, -, -\rangle$ from $\mathcal{H}_i$; $\mathcal{H}_i$.GET(c) to indicate that the protocol retrieves a tuple associated with the command c in $\mathcal{H}_i$; and $\mathcal{H}_i$.GETPREDECESSORS(c) to indicate that the protocol retrieves the set $\mathcal{P}red$ of a tuple $\langle c, -, \mathcal{P}red, -, -, -\rangle$ in $\mathcal{H}_i$. The initial value of $\mathcal{H}_i$ is an empty map.

$\underline{\mathcal{B}allots_i}$. It is an array mapping commands to ballots, which have values in $\mathbb{N}$. $\mathcal{B}allots_i[c] = \mathcal{B}$ means that $\mathcal{B}$ is the current ballot for which p_i has processed an event related to command c. The initial values of $\mathcal{B}allots_i$ are 0.

3.4.2 *Fast Decision*

A client proposes a command c by triggering the event PROPOSE(c) on one of the nodes of CAESAR (lines I1–I2), which becomes c's leader. Let us call this node p_i. p_i enters the *fast proposal phase* for c by choosing the current value of $\mathcal{TS}_i$ as timestamp $\mathcal{T}ime$ of c. The other parameters of this phase are the ballot number $\mathcal{B}allot$ and the whitelist $\mathcal{W}hitelist$ whose values, in this case, are 0 and empty set, respectively. The meaning of these parameters is strictly related to the recovery procedure due to node failures, and therefore we will provide further details in Section 3.4.5. However, at this stage, it is enough to know that:

- a ballot number for c is an identifier of the current leader for c, and a node p_j receiving a message with ballot number $\mathcal{B}$ can process that message only if its current ballot, i.e., $\mathcal{B}allots_j[c]$, for c is not greater than $\mathcal{B}$.
- $\mathcal{W}hitelist$ for c contains the commands that should be considered as predecessors of c according to the perception of the node that is executing a recovery procedure for c.

Fast proposal phase. The purpose of the *fast proposal phase* for a command c with a timestamp $\mathcal{T}ime$ is to propose, to a quorum of nodes, the acceptance of c at $\mathcal{T}ime$ and collect, from that quorum, the known predecessor set $\mathcal{P}red$ of commands $\bar{c}$ that should be decided before c at a timestamp less than $\mathcal{T}ime$. To do so, p_i broadcasts a FASTPROPOSE message with c and $\mathcal{T}ime$, and it collects FASTPROPOSER messages from a quorum of nodes (lines P1–P2).

When a node p_j receives a FASTPROPOSE message with c and $\mathcal{T}ime$, it computes the predecessor set $\mathcal{P}red_j$ by calling the COMPUTEPREDECESSORS function (line P13) and updates the entry for c in $\mathcal{H}_j$ by marking that as $fast\text{-}pending$ with $\mathcal{T}ime$ and $\mathcal{P}red_j$ (line P14), and it calls the function WAIT (line P15) to check the wait condition, as described in Section 3.2. p_j also stores in H_j whether the value of $\mathcal{W}hitelist$ is different from null or not (line P14).

A FASTPROPOSER message for c from a node p_j contains a timestamp $\mathcal{T}ime_j$ and a predecessor set $\mathcal{P}red_j$, and it can be marked with either $\mathcal{OK}$ or $\mathcal{NACK}$. If the message is

```
1: function Set ComputePredecessors($c$, $\mathcal{T}ime$, $\mathcal{W}hitelist$)
2:     return $\{\bar{c} : \bar{c} \sim c \wedge$
       $(\mathcal{W}hitelist = null \Rightarrow \exists\langle \bar{c}, \bar{\mathcal{T}}, -, -, -, - \rangle \in \mathcal{H}_j : \bar{\mathcal{T}} < \mathcal{T}ime) \wedge$
       $(\mathcal{W}hitelist \neq null \Rightarrow \bar{c} \in \mathcal{W}hitelist \vee \exists\langle \bar{c}, \bar{\mathcal{T}}, -,$
           $slow\text{-}pending/accepted/stable, -, - \rangle \in \mathcal{H}_j : \bar{\mathcal{T}} < \mathcal{T}ime) \}$
3: function Boolean Wait($c$, $\mathcal{T}ime$)
4:     wait until $\forall\langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, -, -, - \rangle \in \mathcal{H}_j, (\bar{c} \sim c \wedge \mathcal{T}ime < \bar{\mathcal{T}} \wedge$
       $c \notin \overline{\mathcal{P}red} \Rightarrow \exists\langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, accepted/stable, -, - \rangle \in \mathcal{H}_j)$
5:     if $\exists\langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, accepted/stable, -, - \rangle \in \mathcal{H}_j :$
                                   $\bar{c} \sim c \wedge \mathcal{T}ime < \bar{\mathcal{T}} \wedge c \notin \overline{\mathcal{P}red}$ then
6:         return $\mathcal{NACK}$
7:     else return $\mathcal{OK}$
8: function BreakLoop($c$)
9:     $\langle c, \mathcal{T}, \mathcal{P}red, stable, \mathcal{B}, \perp \rangle \leftarrow \mathcal{H}_j$.Get($c$)
10:    for all $\bar{c} \in \mathcal{P}red : \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, stable, \overline{\mathcal{B}}, \perp \rangle \in \mathcal{H}_j \wedge \bar{\mathcal{T}} < \mathcal{T}$ do
11:        $\mathcal{H}_j$.Update($\bar{c}$, $\bar{\mathcal{T}}$, $\overline{\mathcal{P}red} \setminus \{c\}$, $stable$, $\overline{\mathcal{B}}$, $\perp$)
12:    for all $\bar{c} \in \mathcal{P}red : \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, stable, \overline{\mathcal{B}}, \perp \rangle \in \mathcal{H}_j \wedge \bar{\mathcal{T}} > \mathcal{T}$ do
13:        $\mathcal{P}red \leftarrow \mathcal{P}red \setminus \{\bar{c}\}$
14:    $\mathcal{H}_j$.Update($c$, $\mathcal{T}$, $\mathcal{P}red$, $stable$, $\mathcal{B}$, $\perp$)
15: function Boolean Deliverable($c$)
16:    return $(c \cup \mathcal{H}_j$.GetPredecessors$(c)) \subseteq \mathcal{D}ecided_j$
```

Fig. 3. Auxiliary functions - node p_j

marked with $\mathcal{OK}$, then $\mathcal{T}ime_j$ is equal to the proposed $\mathcal{T}ime$, by meaning that p_j did not reject $\mathcal{T}ime$. On the contrary, if the message is marked with $\mathcal{NACK}$, then $\mathcal{T}ime_j$ is greater than $\mathcal{T}ime$ meaning that p_j rejected $\mathcal{T}ime$ and suggested a greater timestamp for c. In both cases, whether $\mathcal{T}ime$ has been rejected or not, the predecessor set $\mathcal{P}red_j$ contains all the commands $\bar{c}$ that should be decided before c according to the current knowledge of p_j.

Wait (see lines 3–7 of Figure 3) forces c to wait for any command $\bar{c}$ in $\mathcal{H}_j$ that does not commute with c to be marked with either $accepted$ or $stable$, if $\bar{c}$'s timestamp is greater than c's timestamp and c is not in $\bar{c}$'s predecessor set. Afterwards, when the wait condition does not hold anymore, Wait returns $\mathcal{NACK}$ in case there still exists such a command $\bar{c}$, with status either $accepted$ or $stable$; otherwise the function returns $\mathcal{OK}$.

If Wait returns $\mathcal{OK}$, then p_j sends $\mathcal{T}ime$ and the computed $\mathcal{P}red_j$ back to c's leader by confirming what the leader proposed (line P20). Otherwise, if Wait returns $\mathcal{NACK}$ (lines P16–P20), p_j rejects the proposed timestamp by: marking the tuple of c in $\mathcal{H}_j$ as $rejected$, suggesting the current value of $\mathcal{TS}_j$ as a new timestamp for c, and recomputing the predecessor set according to the new timestamp.

The predecessor set $\mathcal{P}red_j$ of c is computed as the set of commands $\bar{c}$ in H_j that do not commute with c and have a timestamp smaller than c's timestamp, with the following exception (see lines 1–2 of Figure 3): if the $\mathcal{W}hitelist$ in input is not null and $\bar{c}$ is not contained in $\mathcal{W}hitelist$, then $\bar{c}$ has to appear with a status that is different from $fast\text{-}pending$ in $\mathcal{H}_j$ in order to be included in $\mathcal{P}red_j$.

In case of a *fast decision* (see *FastDecision* transition in Figure 4), the command leader p_i is able to collect a fast quorum of $\mathcal{FQ}$ replies that do not reject $\mathcal{T}ime$ for c (line P5). It then submits c with the confirmed $\mathcal{T}ime$ and the union of the received predecessor sets, i.e., $\mathcal{P}red$, to the next *stable phase* (lines P3–P4 and P6).

Note that unlike other multi-leader consensus protocols [10], [13], a fast decision in Caesar is guaranteed in case a fast quorum confirms the timestamp for a command, although those nodes can reply with non-equal predecessors sets. In the correctness proof of Caesar (see Section 3.4.6), we show that such a condition is sufficient to guarantee the recoverability of the fast decision for c even in case the command leader and at most other $f-1$ nodes crash.

Stable phase. The purpose of the *stable phase* for a command c with a timestamp $\mathcal{T}ime$ and predecessor set $\mathcal{P}red$ is to communicate to all the nodes, via a Stable message, that c has to be decided at timestamp $\mathcal{T}ime$ after all the commands in $\mathcal{P}red$ have been decided (line S1). In particular, whenever a node p_j receives a Stable message for c, with $\mathcal{T}ime$ and set $\mathcal{P}red$ (lines S2–S7), it updates the tuple for c in $\mathcal{H}_j$ with the new values and marks the tuple as $stable$ (line S3).

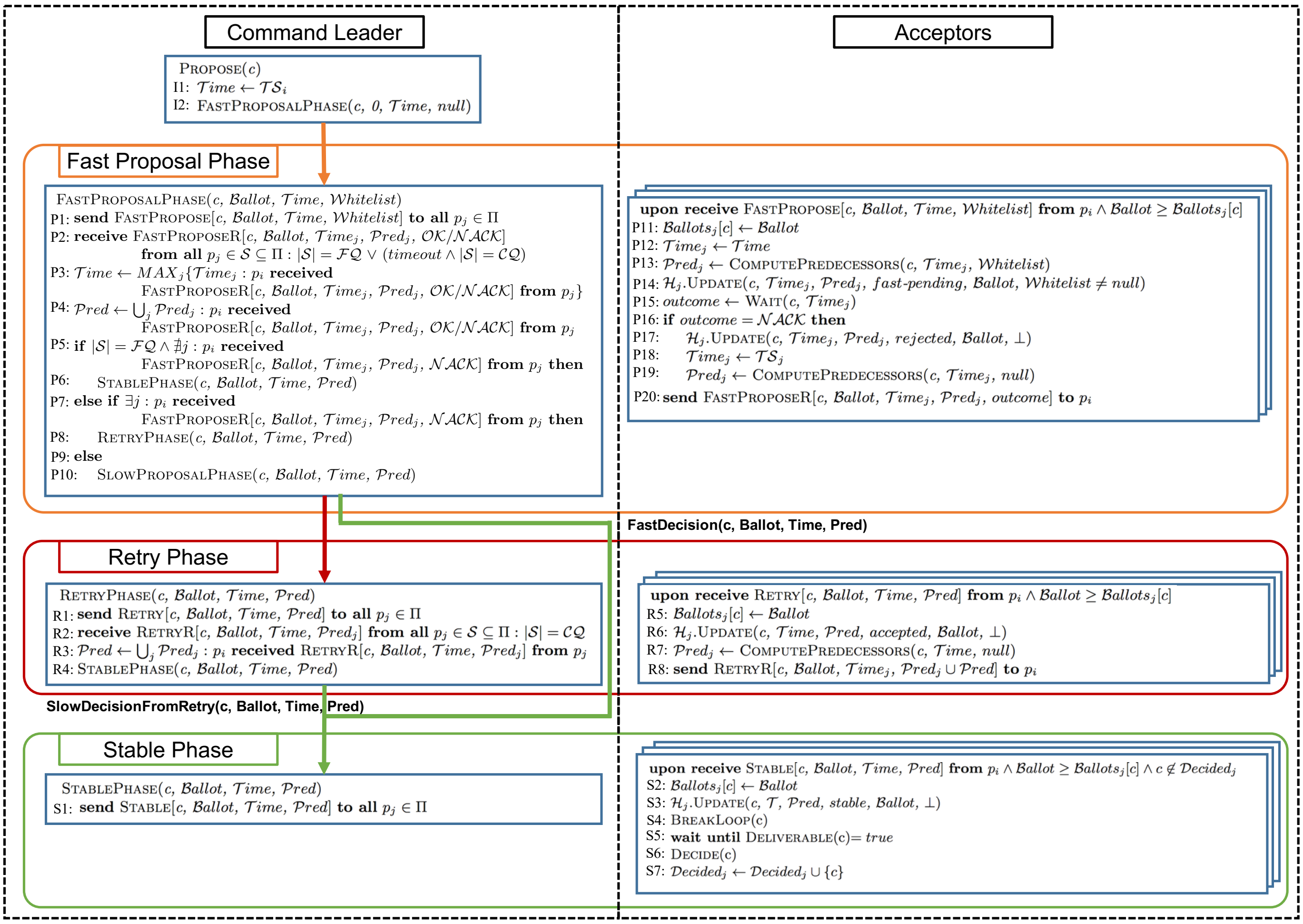
Whenever each command in $\mathcal{P}red$ has been decided (lines 16–17 of Figure 3), p_j can decide c by triggering Decide(c) (lines S5–S7). This is correct because, as we prove in Section 3.4.6, the phases executed before the stable phase guarantee that for any pair of $stable$ and non-commutative commands c and $\bar{c}$, with timestamps $\mathcal{T}ime$ and $\overline{\mathcal{T}ime}$ respectively, if $\overline{\mathcal{T}ime} < \mathcal{T}ime$ then $\bar{c} \in \mathcal{P}red$, where $\mathcal{P}red$ is the predecessor set of c. Therefore, the decision order of non-commutative commands is guaranteed to follow the increasing order of the commands' timestamps. However, this does not mean that if $\bar{c} \in \mathcal{P}red$, then $\overline{\mathcal{T}ime} < \mathcal{T}ime$. Hence the stable phase has to take care of breaking any possible loop that might be created by the predecessor sets of the $stable$ commands, before trying to deliver them (line S4 and lines 9–15 of Figure 3). That is done as follows: for any two $stable$ and non-commutative commands c and $\bar{c}$ with timestamps $\mathcal{T}$ and $\bar{\mathcal{T}}$, respectively, if $\bar{\mathcal{T}} > \mathcal{T}$ then $\bar{c}$ is deleted from c's predecessor set.

When a command c is stable on all nodes, the information about c can be safely garbage collected.

3.4.3 Slow Decision

In case the leader of a command c cannot guarantee a fast decision for c, then it has to execute additional phases before the finalization of the *stable phase* for c. This happens because in the *fast proposal phase* for c (lines I1–I2, P1–P4, and P11–P20), the command leader cannot collect a fast quorum of FastProposeR messages that are all marked with $\mathcal{OK}$ (lines P7–P10) due to the following reasons: the fast quorum of collected FastProposeR messages actually includes a message that rejects the proposed timestamp for c and is marked with $\mathcal{NACK}$ (lines P7–P8, and R1–R8); or the leader is only able to collect a quorum of $\mathcal{CQ}$ FastProposeR messages (lines P9–P10), because either there are no $\mathcal{FQ}$ correct nodes in the system or the other $N - \mathcal{CQ}$ nodes are too slow to provide their reply within a configurable timeout to the command leader (line P2). In this subsection, we refer to a *slow decision* by focusing on the former case; the latter is explained in Section 3.4.4.

Retry phase. This phase guarantees that a command c is accepted by a quorum of $\mathcal{CQ}$ nodes after the previous *proposal phase* for c could not provide a fast decision, and before moving to the *stable phase* for c. At this stage, the leader p_i of c broadcasts a Retry message with the maximum $\mathcal{T}ime$ among the ones suggested by the acceptors in the previous phase, and the predecessor set $\mathcal{P}red$ as the union of the sets suggested by the acceptors in the previous phase (line R1). Then p_i waits for a quorum of $\mathcal{CQ}$ RetryR replies that confirm the timestamp $\mathcal{T}ime$ for c (line R2), before submitting $\mathcal{T}ime$ to the next *stable phase* (line R4). This


Command Leader
Acceptors
Fast Proposal Phase
FastDecision(c, Ballot, Time, Pred)
Retry Phase
SlowDecisionFromRetry(c, Ballot, Time, Pred)
Stable Phase

```
PROPOSE($c$)
I1: $Time \leftarrow TS_i$
I2: FASTPROPOSALPHASE($c$, $0$, $Time$, $null$)
```

```
FASTPROPOSALPHASE($c$, $Ballot$, $Time$, $Whitelist$)
P1: send FASTPROPOSE[$c$, $Ballot$, $Time$, $Whitelist$] to all $p_j \in \Pi$
P2: receive FASTPROPOSER[$c$, $Ballot$, $Time_j$, $Pred_j$, $OK/NACK$]
            from all $p_j \in \mathcal{S} \subseteq \Pi : |\mathcal{S}| = \mathcal{FQ} \vee (timeout \wedge |\mathcal{S}| = \mathcal{CQ})$
P3: $Time \leftarrow MAX_j\{Time_j : p_i$ received
            FASTPROPOSER[$c$, $Ballot$, $Time_j$, $Pred_j$, $OK/NACK$] from $p_j\}$
P4: $Pred \leftarrow \bigcup_j Pred_j : p_i$ received
            FASTPROPOSER[$c$, $Ballot$, $Time_j$, $Pred_j$, $OK/NACK$] from $p_j$
P5: if $|\mathcal{S}| = \mathcal{FQ} \wedge \nexists j : p_i$ received
            FASTPROPOSER[$c$, $Ballot$, $Time_j$, $Pred_j$, $NACK$] from $p_j$ then
P6:     STABLEPHASE($c$, $Ballot$, $Time$, $Pred$)
P7: else if $\exists j : p_i$ received
            FASTPROPOSER[$c$, $Ballot$, $Time_j$, $Pred_j$, $NACK$] from $p_j$ then
P8:     RETRYPHASE($c$, $Ballot$, $Time$, $Pred$)
P9: else
P10:    SLOWPROPOSALPHASE($c$, $Ballot$, $Time$, $Pred$)
```

```
upon receive FASTPROPOSE[$c$, $Ballot$, $Time$, $Whitelist$] from $p_i \wedge Ballot \geq Ballots_j[c]$
P11: $Ballots_j[c] \leftarrow Ballot$
P12: $Time_j \leftarrow Time$
P13: $Pred_j \leftarrow$ COMPUTEPREDECESSORS($c$, $Time_j$, $Whitelist$)
P14: $\mathcal{H}_j$.UPDATE($c$, $Time_j$, $Pred_j$, $fast\text{-}pending$, $Ballot$, $Whitelist \neq null$)
P15: $outcome \leftarrow$ WAIT($c$, $Time_j$)
P16: if $outcome = NACK$ then
P17:     $\mathcal{H}_j$.UPDATE($c$, $Time_j$, $Pred_j$, $rejected$, $Ballot$, $\perp$)
P18:     $Time_j \leftarrow TS_j$
P19:     $Pred_j \leftarrow$ COMPUTEPREDECESSORS($c$, $Time_j$, $null$)
P20: send FASTPROPOSER[$c$, $Ballot$, $Time_j$, $Pred_j$, $outcome$] to $p_i$
```

```
RETRYPHASE($c$, $Ballot$, $Time$, $Pred$)
R1: send RETRY[$c$, $Ballot$, $Time$, $Pred$] to all $p_j \in \Pi$
R2: receive RETRYR[$c$, $Ballot$, $Time$, $Pred_j$] from all $p_j \in \mathcal{S} \subseteq \Pi : |\mathcal{S}| = \mathcal{CQ}$
R3: $Pred \leftarrow \bigcup_j Pred_j : p_i$ received RETRYR[$c$, $Ballot$, $Time$, $Pred_j$] from $p_j$
R4: STABLEPHASE($c$, $Ballot$, $Time$, $Pred$)
```

```
upon receive RETRY[$c$, $Ballot$, $Time$, $Pred$] from $p_i \wedge Ballot \geq Ballots_j[c]$
R5: $Ballots_j[c] \leftarrow Ballot$
R6: $\mathcal{H}_j$.UPDATE($c$, $Time$, $Pred$, $accepted$, $Ballot$, $\perp$)
R7: $Pred_j \leftarrow$ COMPUTEPREDECESSORS($c$, $Time$, $null$)
R8: send RETRYR[$c$, $Ballot$, $Time_j$, $Pred_j \cup Pred$] to $p_i$
```

```
STABLEPHASE($c$, $Ballot$, $Time$, $Pred$)
S1: send STABLE[$c$, $Ballot$, $Time$, $Pred$] to all $p_j \in \Pi$
```

```
upon receive STABLE[$c$, $Ballot$, $Time$, $Pred$] from $p_i \wedge Ballot \geq Ballots_j[c] \wedge c \notin Decided_j$
S2: $Ballots_j[c] \leftarrow Ballot$
S3: $\mathcal{H}_j$.UPDATE($c$, $\mathcal{T}$, $Pred$, $stable$, $Ballot$, $\perp$)
S4: BREAKLOOP(c)
S5: wait until DELIVERABLE(c)= $true$
S6: DECIDE(c)
S7: $Decided_j \leftarrow Decided_j \cup \{c\}$
```

Fig. 4. CAESAR's pseudocode. The left part is executed by the command c's leader p_i, and the right part can be executed by any acceptor p_j (including p_i).

guarantees that, even with f failures, there always exists a correct node that confirmed $Time$ in this phase.

It is important to notice that as in the case of a FASTPROPOSER message, a RETRYR message from a node p_j also contains p_j's view of c's predecessors set, which will be included in the final $Pred$ set in input to the next *stable phase* (line R3). This is because, as shown in Section 3.3, c's leader has to include all the commands that were not predecessors of c according to the timestamp proposed in the previous *proposal phase* but that have to be considered as predecessors according to the new timestamp of this phase.

Furthermore, a reply from an acceptor in this phase *cannot reject* the broadcast timestamp for c, because, as it will be clear in the proof of correctness (see Section 3.4.6), at this stage CAESAR guarantees that there does not exist any acceptor p_j and command $\bar{c}$ such that $\bar{c}$ is *stable* on p_j with timestamp $\bar{\mathcal{T}} > \mathcal{T}$ and c is not in $\bar{c}$'s predecessors set. Therefore, when a node p_j receives a RETRYR message with c, $Time$, and $Pred$, it only updates the tuple for c in its $\mathcal{H}_j$ by marking it as *accepted* with $Time$ and $Pred$ (line R5), and it computes a new predecessors set $Pred_j$ by calling the COMPUTEPREDECESSORS function (line R7), like in the *fast proposal phase*. Then, it sends a confirmation RETRYR back to the command leader with the new $Pred_j$ as well as the one previously received by the leader (line R8).

3.4.4 Unavailability of Fast Quorums

In CAESAR, as in other fast consensus implementations [10], there might exist scenarios where no fast quorum is available. This happens due to our choice on the size of fast quorums, i.e., $\mathcal{FQ}$, which is greater than the minimum number of correct nodes in the system, i.e., $N - f$. Therefore, under a period of asynchrony of the system, where a message can experience an arbitrarily long delay, a node is not able to distinguish whether f nodes crashed or not, and hence a command leader that waits for replies from a fast quorum of nodes could wait indefinitely in a *fast proposal phase*.

This issue is solved in CAESAR by adopting a more common solution, namely the adoption of timeouts, but it requires the interposition of an additional *slow proposal phase* after the *fast proposal phase* and before either the *retry* or the *stable phase*. In particular, a command leader can decide to execute a *slow proposal phase* without waiting for a fast quorum of $\mathcal{FQ}$ replies if it has collected a quorum of $\mathcal{CQ}$ FASTPROPOSER messages for a command c and none of the messages have rejected the proposed timestamp (P9–P10).This scenario can be considered as a corner case of CAESAR's execution and thus, for the sake of brevity, we decided to detail it in the technical report [20].

3.4.5 Recovery from Failures

Whenever a node p_i crashes, there might exist some command c whose leader is p_i and whose decision would never be finalized unless some explicit action is taken. Indeed, let us suppose there exists a node p_k that stores c with a status different from $stable$. Then, according to the pseudocode of Figure 4, p_k would decide c only after having received a STABLE message from p_i.

```
1:  RECOVERYPHASE(c)
2:      $\mathcal{B}allots_k[c]$++
3:      send RECOVERY[c, $\mathcal{B}allots_k[c]$] to all $p_j \in \Pi$
4:      receive RECOVERYR[c, $\mathcal{B}allots_k[c]$, $\langle c, \mathcal{T}_j, \mathcal{P}red_j, -, \mathcal{B}_j, \perp/\top \rangle /$
    $\mathcal{NOP}$] from all $p_j \in \mathcal{S} \subseteq \Pi : |\mathcal{S}| = \mathcal{CQ}$
5:      $\mathcal{M}axBallot \leftarrow MAX\{\mathcal{B}_j : p_i$ received
                RECOVERYR[c, $\mathcal{B}allots_k[c]$, $\langle c, \mathcal{T}_j, \mathcal{P}red_j, -, \mathcal{B}_j, \perp/\top \rangle$] $\}$
6:      $\mathcal{R}ecoverySet \leftarrow \{\langle p_j, \mathcal{T}_j, \mathcal{P}red_j, -, \perp/\top \rangle : p_i$ received
                RECOVERYR[c, $\mathcal{B}allots_k[c]$, $\langle c, \mathcal{T}_j, \mathcal{P}red_j, -, \mathcal{B}_j, \perp/\top \rangle$]
                                        from $p_j \wedge \mathcal{B}_j = \mathcal{M}axBallot$ $\}$
7:      if $\exists \langle p_j, \mathcal{T}_j, \mathcal{P}red_j, stable, \perp \rangle \in \mathcal{R}ecoverySet$ then
8:          STABLEPHASE(c, $\mathcal{B}allots_k[c]$, $\mathcal{T}_j$, $\mathcal{P}red_j$)
9:      else if $\exists \langle p_j, \mathcal{T}_j, \mathcal{P}red_j, accepted, \perp \rangle \in \mathcal{R}ecoverySet$ then
10:         RETRYPHASE(c, $\mathcal{B}allots_k[c]$, $\mathcal{T}_j$, $\mathcal{P}red_j$)
11:     else if $\exists \langle p_j, \mathcal{T}_j, \mathcal{P}red_j, rejected, \perp \rangle \in \mathcal{R}ecoverySet$ then
12:         $\mathcal{T}ime \leftarrow \mathcal{TS}_i$
13:         FASTPROPOSALPHASE(c, $\mathcal{B}allots_k[c]$, $\mathcal{T}ime$, $null$)
14:     else if $\exists \langle p_j, \mathcal{T}_j, \mathcal{P}red_j, slow\text{-}pending, \perp \rangle \in \mathcal{R}ecoverySet$ then
15:         SLOWPROPOSALPHASE(c, $\mathcal{B}allots_k[c]$, $\mathcal{T}_j$, $\mathcal{P}red_j$)
16:     else if $|\mathcal{R}ecoverySet| > 0$ then
17:         $\mathcal{T}ime \leftarrow \mathcal{T}_j$ :
                $\exists \langle p_j, \mathcal{T}_j, \mathcal{P}red_j, fast\text{-}pending, \perp/\top \rangle \in \mathcal{R}ecoverySet$
18:         $\mathcal{P}red \leftarrow \bigcup_j \mathcal{P}red_j$ :
                $\langle p_j, \mathcal{T}_j, \mathcal{P}red_j, fast\text{-}pending, \perp/\top \rangle \in \mathcal{R}ecoverySet$
19:         if $\exists \langle p_j, \mathcal{T}_j, \mathcal{P}red_j, fast\text{-}pending, \top \rangle \in \mathcal{R}ecoverySet$ then
20:             $\mathcal{W}hiteList \leftarrow \mathcal{P}red$
21:         else if $|\mathcal{R}ecoverySet| \geq \lfloor \frac{\mathcal{CQ}}{2} \rfloor + 1$ then
22:             $\mathcal{W}hiteList \leftarrow \{\bar{c} \in \mathcal{P}red : \nexists \mathcal{S} \subseteq \mathcal{R}ecoverySet, |\mathcal{S}| \geq$
    $\lfloor \frac{\mathcal{CQ}}{2} \rfloor + 1 \wedge \forall \langle p_j, \mathcal{T}_j, \mathcal{P}red_j, fast\text{-}pending, \perp \rangle \in \mathcal{S}, \bar{c} \notin \mathcal{P}red_j \}$
23:         else
24:             $\mathcal{W}hiteList \leftarrow null$
25:         FASTPROPOSALPHASE(c, $\mathcal{B}allots_k[c]$, $\mathcal{T}ime$, $\mathcal{W}hiteList$)
26:     else
27:         $\mathcal{T}ime \leftarrow \mathcal{TS}_i$
28:         FASTPROPOSALPHASE(c, $\mathcal{B}allots_k[c]$, $\mathcal{T}ime$, $null$)
29: upon receive RECOVERY[c, $\mathcal{B}allot$] from $p_k \wedge \mathcal{B}allot > \mathcal{B}allots_j[c]$
30:     $\mathcal{B}allots_j[c] \leftarrow \mathcal{B}allot$
31:     if $\mathcal{H}_j$.CONTAINS(c) then
32:         send RECOVERYR[c, $\mathcal{B}allots_j[c]$, $\mathcal{H}_j$.GETINFO(c)] to $p_k$
33:     else
34:         send RECOVERYR[c, $\mathcal{B}allots_j[c]$, $\mathcal{NOP}$] to $p_k$
```

Fig. 5. RECOVERY phase executed by node p_k. Node p_j is a receiver of the RECOVERY message.

For this reason, CAESAR also includes an explicit recovery procedure (Figure 5) that finalizes the decision of commands whose leader either crashed or has been suspected. Recall that a failure detector [17] exists at each node to detect node failures. Given the aforementioned example, whenever the failure detector of p_k suspects p_i, p_k attempts to become c's leader and finalizes the decision of c. This is done by executing a Paxos-like prepare phase, and collecting the most recent information about c from a quorum of $\mathcal{CQ}$ nodes as follows: p_k increments its current ballot for c, i.e., $\mathcal{B}allots_k[c]$, (line 2) and it broadcasts a RECOVERY message for c with the new ballot (line 3). Then, it waits for a quorum of $\mathcal{CQ}$ RECOVERYR replies, which contain information about c, before finalizing the decision for c (line 4). RECOVERYR from p_j contains either the tuple of c in $\mathcal{H}_j$ or $\mathcal{NOP}$ if such a tuple does not exist (lines 31–34).

A node p_j that receives a RECOVERY message from p_k replies only if its ballot for c is lesser than the one it has received. In such a case, p_j also updates its ballot for c (lines 29–30). Like in Paxos, this is done to guarantee that no two leaders can compete to finalize the decision for the same command concurrently. In fact, if two leaders p_{k1} and p_{k2} both successfully execute lines 3 and 4 of the recovery procedure with ballots $\mathcal{B}_1$ and $\mathcal{B}_2$, respectively, then, if $\mathcal{B}_1 < \mathcal{B}_2$, for any quorum of nodes $\mathcal{S}$, there always exists a node in $\mathcal{S}$ that never replies to p_{k1} (see the reception of FASTPROPOSE, SLOWPROPOSE, RETRY, and STABLE messages in Figure 4).

When node p_k successfully becomes c's leader, it filters the information for c that it has received by only keeping in $\mathcal{R}ecoverySet$ the data associated with the maximum ballot, named $\mathcal{M}axBallot$ in the pseudocode (lines 5–6). Each tuple of the set is a sequence of *node identifier*, *timestamp*, *predecessors set*, *status*, and *forced boolean* indicating: the node that sent the information, the timestamp, the predecessors set, the status of c on that node, and whether that information has been forced by a $\mathcal{W}hiteList$ or not on that node. Then, p_k takes a decision for c according to the content of $\mathcal{R}ecoverySet$ as follows. *i)* If there exists a tuple with status $stable$, then p_k starts a *stable phase* for c by using the necessary info from that tuple, e.g., timestamp and predecessors set (lines 7–8). *ii)* If there exists a tuple with status $accepted$, then p_k starts a *retry phase* for c by using the necessary info from that tuple (lines 9–10). *iii)* If there exists a tuple with status $rejected$ or $\mathcal{R}ecoverySet$ is empty, c was never decided, and hence p_k starts a *fast proposal phase* for c (lines 11–13, and 26–28) by using a new timestamp (as described in Section 3.4.2). *iv)* If there exists a tuple with status $slow\text{-}pending$, then p_k starts a *slow proposal phase* for c by using the necessary info from that tuple (lines 14–15). *v)* If the previous conditions are false, then $\mathcal{R}ecoverySet$ contains tuples with the same timestamp $\mathcal{T}ime$ and status $fast\text{-}pending$ (lines 16–25). In this last case, p_k starts a *proposal phase* for c with timestamp $\mathcal{T}ime$ because c might have been decided with that timestamp in a previous fast decision (line 25). If so, p_k has to also choose the right predecessors set that was adopted in that decision. Therefore, it has to either choose a predecessors set in $\mathcal{R}ecoverySet$ that was forced by a previous recovery, if any (lines 19–20), or it has to build its own $\mathcal{W}hiteList$ of commands that should be forced as predecessors of c (lines 21–24).

This is done by noticing that: if c was decided in a *fast decision* with ballot $\mathcal{M}axBallot$ then the size of $\mathcal{R}ecoverySet$ cannot be lesser than $\lfloor \frac{\mathcal{CQ}}{2} \rfloor + 1$, which is the minimum size of the intersection of any classic quorum and any fast quorum (lines 21 and 24); if a command $\bar{c}$ was previously decided in a *fast decision* and it has to be a predecessor of c, then there cannot exist a subset of $\lfloor \frac{\mathcal{CQ}}{2} \rfloor + 1$ tuples in $\mathcal{R}ecoverySet$, whose predecessors sets do not contain $\bar{c}$ (line 22). Note that, the case in which $\bar{c}$ was previously decided in a *slow decision* and has to be a predecessor of c is handled by the computation of predecessors set in the *fast proposal phase* (see line P13 of Figure 4, and lines 1–3 of Figure 3).

3.4.6 Correctness

The complete formal proof on the correctness of CAESAR is in the technical report [20], where we also provide a TLA+ specification [21] of the algorithm, which has been model-checked with TLC model-checker. Here, we provide an informal proof to convey the main intuition of our design.

Let us define the predicate DECIDED[c,$\mathcal{T}$,$\mathcal{P}red$,$\mathcal{B}$] that is equal to true whenever a node decides a command c with timestamp $\mathcal{T}$, predecessors set $\mathcal{P}red$, and ballot $\mathcal{B}$. Note that a command is *decided* only when its predecessors with lower timestamps are *decided*, after undergoing the STABLE phase (Figure 4 lines S4–S7). With this, CAESAR's *Consistency* guarantees can be proved using the following two theorems. The first theorem dictates that a command predecessor set is strictly determined by its timestamp, while the second theorem dictates that, once decided, a command will possess the same set of predecessors at all higher ballots.

Theorem 1. *Command Precedence.* $\forall c, \bar{c}$: DECIDED$[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge$ DECIDED$[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \overline{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge c \sim \bar{c} \Rightarrow \bar{c} \in \mathcal{P}red$

Proof Sketch. By contradiction. Suppose that DECIDED$[c,\mathcal{T},\mathcal{P}red,\mathcal{B}] \wedge$ DECIDED$[\bar{c},\bar{\mathcal{T}},\overline{\mathcal{P}red},\overline{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge c \sim \bar{c} \wedge \bar{c} \notin \mathcal{P}red$.

If c undergoes a RETRY *phase*: Suppose that c first selects a timestamp $\mathcal{T}' < \bar{\mathcal{T}}$ (since $\bar{c} \notin \mathcal{P}red$) and then it retries with $\mathcal{T}$ due to a $\mathcal{NACK}$ received in the *Fast Proposal* phase (Figure 4, P7). Since c did not observe $\bar{c}$ and by definition of quorum, the quorum of nodes replying to the proposal of $\bar{c}$ includes c (i.e. $c \in \overline{\mathcal{P}red}$). Therefore, when c undergoes the *Retry* phase with the new $\mathcal{T} > \bar{\mathcal{T}}$, it must be that $\bar{c} \in \mathcal{P}red$ (Figure 3, L2). There should exist at least one node that is in both the quorum that replies in the *Fast Proposal* phase of $\bar{c}$ and the one that replies in the *Retry* phase of c. This contradicts the hypothesis $\bar{c} \notin \mathcal{P}red$.

If c does not undergo a Retry phase: Suppose that c is *stable* with timestamp $\mathcal{T}$ such that $\bar{c} \notin \mathcal{P}red$. Then, there exists at least one node p_j in the intersection of fast and slow quorums that responds with a $\mathcal{NACK}$ in $\bar{c}$'s *fast proposal* phase (Figure 4, Line P15). Thus, $\bar{c}$ will be *retried* at position $\bar{\mathcal{T}} > \mathcal{T}$. This contradicts the hypothesis that $\bar{\mathcal{T}} < \mathcal{T}$. □

Theorem 2. *Recoverability.* $\forall c$: ($\exists \mathcal{B}$: DECIDED$[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge \forall \bar{c} \in \mathcal{P}red$, DECIDED$[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \overline{\mathcal{B}}] \Rightarrow (\forall \mathcal{B}' \geq \mathcal{B}$: DECIDED$[c, \mathcal{T}', \mathcal{P}red', \mathcal{B}'] \Rightarrow \mathcal{T}' = \mathcal{T} \wedge \mathcal{P}red' = \mathcal{P}red$))

Proof Sketch. If DECIDED$[c,\mathcal{T},\mathcal{P}red,\mathcal{B}]$ happened after a *Retry* phase, there should exist a $p_i \in \mathcal{CQ}$ that has *accepted* c. Due to the non-empty intersection of quorums, DECIDED$[\bar{c},\bar{\mathcal{T}},\overline{\mathcal{P}red},\overline{\mathcal{B}}]$ implies that, in the recovery procedure for c at $\mathcal{B}'$, there is at least one node $p_k \in \mathcal{CQ}$ ($p_k \neq p_i$) that will respond to c's recovery and will include $\bar{c}$ in $\mathcal{P}red'$ (Figure 5, L31-32).

If c was decided by a *Stable* immediately after a *Fast Proposal* phase, then there should be at least $\mathcal{FQ}$ nodes where c is either *stable* or *fast-pending*. Depending on how $\bar{c} \in \mathcal{P}red$ is decided, we have two cases. If DECIDED$[\bar{c},\bar{\mathcal{T}},\overline{\mathcal{P}red},\overline{\mathcal{B}}]$ is due to a *stable* after a *fast proposal* phase, then $\bar{c}$ will be *fast-pending* at least in $\mathcal{FQ}$ nodes. Due to the intersection of any two fast quorums as well as a fast and a slow quorum, the recovery procedure will pick up c with $\bar{c} \in \mathcal{P}red$ from at least one node in the intersection. If $\bar{c}$ was decided after a *Retry*, then $\mathcal{CQ}$ nodes should have *accepted/stable* status for $\bar{c}$ in its history. Since any two fast quorums and a slow quorum must intersect, there should at least be one node in the intersection that will respond to c's recovery message. □

The *Consistency* property that no two different nodes decide different values is satisfied by Theorems 1 and 2. The first ensures that for any *decided* command, its predecessor set always includes all conflicting commands with lower timestamp at all nodes, while the second ensures that once *decided*, the command tuple is available in at least one node in any quorum, that the recovery procedure can pickup and enforce at all higher ballots.

3.5 Implementation and Evaluation

We implemented CAESAR in Java and contrasted it with four state-of-the-art consensus protocols: M^2Paxos, EPaxos, Multi-Paxos, and Mencius. We used the Go language implementations of EPaxos, Multi-Paxos, and Mencius from the authors of EPaxos. For M^2Paxos, we used the open-source implementation in Go. Note that Go compiles to native binary while Java runs on top of the Java Virtual Machine. Thus, we use a warmup phase before each experiment in order to kickstart the Java JIT Compiler.

Competitors have been evaluated on Amazon EC2, using m4.2xlarge instances (8 vCPU and 32GB RAM) running Ubuntu Linux 16.04. Our benchmark issues client commands to update a given key of a fully replicated key-value store. Two commands are conflicting if they access the same key. The command size is 15 bytes, which include key, value, request ID, and operation type.

In our evaluations, we explored both conflicting and non-conflicting workloads. When the clients issue conflicting commands, the key is picked from a shared pool of 100 keys with a certain probability depending on the experiment. To measure latency, we issued requests in a closed loop by placing 10 clients co-located with each node (50 in total), and for throughput the clients injected requests to the system in an open loop. Performance of competitors has been collected with and without network batching (indicated by the captions).

The reasons behind using a 100-key conflict pool are as follows. Firstly, fifty geographically distributed clients accessing the same key at the same time does not reflect practical application scenarios. To induce a conflict, it is enough for any two distant clients to access the same key, which our size of conflict pool enables. Secondly, due to the open-loop nature of the throughput experiment, the effective conflict rate usually far exceeds the target rate. Thus, commands fall prey to the wait condition unnecessarily and result in lower performance. The choice of 100-key pool provides a level platform for both the latency and throughput experiments. An additional latency experiment with a smaller conflict pool is presented in Appendix A

We deployed the competitors on five nodes located in Virginia (US), Ohio (US), Frankfurt (EU), Ireland (EU), and Mumbai (India). This configuration spreads nodes such that the latency to achieve a quorum is similar for all quorum-based competitors. It is worth recalling that in a system with 5 nodes, CAESAR requires contacting one node more than other quorum-based competitors to reach a fast decision. The round trip time (RTT) that we measured in between nodes in EU and US are all below 100ms. The node in India experiences the following delays with respect to the other nodes: 186ms/VA, 301ms/OH, 112ms/DE, 122ms/IR. As in EPaxos, CAESAR uses separate queues for handling different types of messages, and each of these queues is handled by a separate pool of threads. CAESAR tracks conflicting commands using a Red-Black tree data structure ordered by

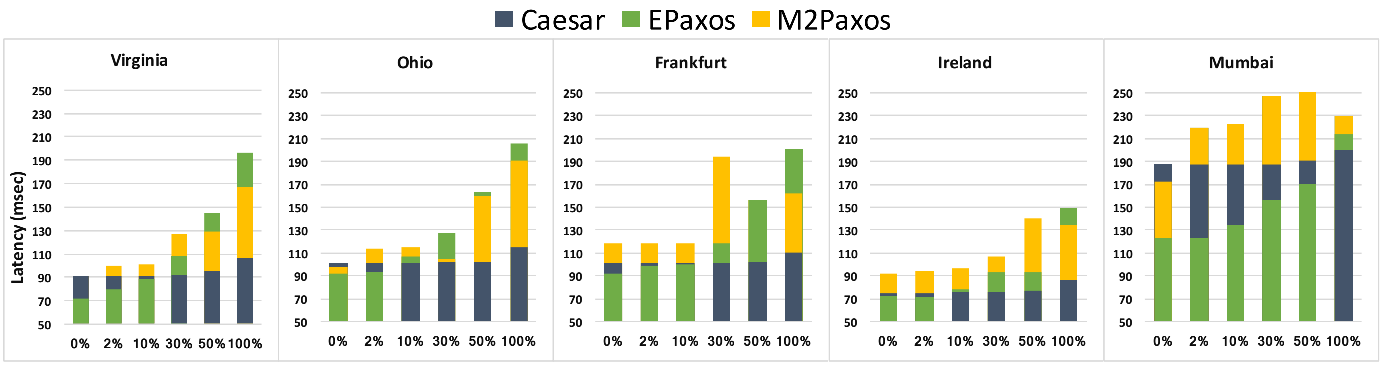


Fig. 6. Average latency for ordering and processing commands by varying the percentage of commands picked from the 100-key conflict pool. Batching is disabled. Bars are overlapped: e.g., in the case of 30% in Virginia, latency values are 90 msec, 108 msec, and 127 msec, for CAESAR, EPaxos, and M^2Paxos, respectively.

their timestamp. Further, Multi-Paxos is deployed in two settings: one where the leader is located in Ireland, which is a node close to a quorum, and one where the leader is in Mumbai, which needs to contact nodes at long distances to receive a quorum of responses.

3.5.1 *Non-faulty Scenarios*

In Figure 6, we report the average latency incurred by CAESAR, EPaxos, and M^2Paxos to order and execute a command. Given the latency of a command is affected by the position of the leader that proposes the command itself, we show the results collected at each site. Each cluster of data shows the behavior of a system while increasing the percentage of commands picked from the conflict pool in the range of {0% – no conflict, 2%, 10%, 30%, 50%, 100%}.

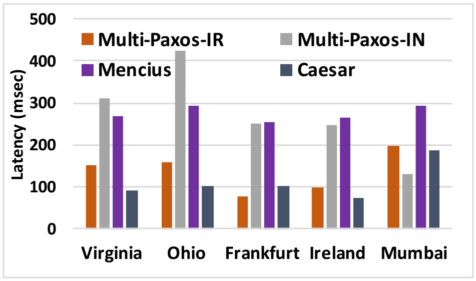


Fig. 7. Average latency for ordering commands of Multi-Paxos (with a close and faraway leader), Mencius, and CAESAR. Batching is disabled.

At 0% conflicts, EPaxos and M^2Paxos provide comparable performance because both employ two communication steps to order commands and the same size for quorums, with EPaxos slightly faster because it does not need to acquire the ownership on submitted commands before ordering. The performance of CAESAR is slightly slower (on average 18%) than EPaxos because of the need of contacting one more node to reach consensus.

When the percentage of commands picked from the conflict pool increases up to 50%, CAESAR sustains its performance by providing an almost constant latency; all other competitors degrade their performance visibly. The reasons vary by protocol. EPaxos degrades because its number of slow decisions increases accordingly, along with the complexity of analyzing the conflict graph before delivering. For M^2Paxos, the degradation is related to the forwarding mechanism implemented when the requested key is logically owned by another node. In that case, M^2Paxos passes the command to that node, which becomes responsible to order it. This mechanism introduces an additional communication delay, which contributes to degraded performance especially in geo-scale where the node having the ownership of the key may be faraway. At last, we also included the case where 100% of commands are from the conflict pool. Here all competitors behave poorly given the need for ordering all commands, which does not represent their ideal deployment.

The latency provided by the node in India is higher than other nodes. Here CAESAR is 50% slower than EPaxos only when conflicts are low, because CAESAR has to contact one more faraway node (e.g., Virginia) to deliver fast.

Performances of Multi-Paxos and Mencius are reported in Figure 7 because these competitors are oblivious to the amount of conflicting commands injected in the system. CAESAR 0% has been included as a reference. Mencius's performance is similar across the nodes because it needs to collect feedback from all consensus participants; therefore, it performs as the slowest node and on average 60% slower than CAESAR. The version of Multi-Paxos with the leader in Mumbai (Multi-Paxos-IN) is not able to provide low latency due to the delay that commands experience while waiting for a response from the leader. On the other hand, if the leader is placed in Ireland (Multi-Paxos-IR) the quorum can be reached faster than the case of Multi-Paxos-IN, thus command latency is significantly lower. Compared to results in Figure 6, Multi-Paxos-IR and Multi-Paxos-IN are, on average, 5% and 40% slower than CAESAR 100%, respectively.

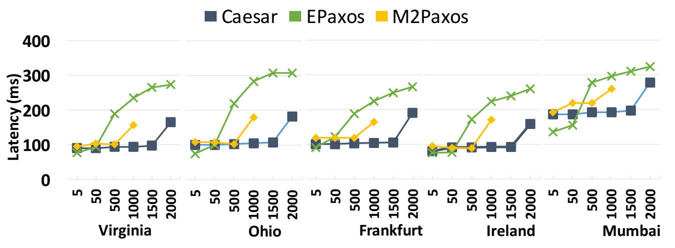


Fig. 8. Latency per node while varying the number of connected clients (5 – 2000). Network messages are not batched.

Scalability of competitors is measured by loading the system with more clients. Figure 8 shows the latency of CAESAR, EPaxos, and M^2Paxos for each site using a workload with 10% *conflict pool*-picked commands. The x-axis indicates the total number of connected clients. The complex delivery phase of EPaxos, where it analyzes the dependency graph before executing every command, slows down its performance as the load increases while CAESAR provides a steady latency and reaches its saturation only when over 1500 total clients are connected. M^2Paxos stops scaling beyond 1000 connected clients due to the impact of the forwarding mechanism.

Figure 9 shows the total throughput obtained by each competitor. Performance of Multi-Paxos and Mencius is placed under the 0% case. Here the performance of CAESAR degrades by only 17% when moving from a no-conflict workload to one with 10% *conflict pool*-picked commands. EPaxos and M^2Paxos have already lost 24% and 45% of their performance with respect to the no-conflict configuration. The cases of 30% and 50% still show improvement for CAESAR, but now the impact of the wait condition to deliver fast is more evident, which explains the gap in throughput from the case of CAESAR 10%. M^2Paxos is the system that behaves best when 100% commands are from the conflict pool. Here the impact of the forwarding technique deployed when commands access an object owned by a different node prevails over the ordering procedure of EPaxos and CAESAR, which involves the exchange of a long list of dependent commands over the network. Interestingly, Multi-Paxos-IR performs as EPaxos 0%. That is because in this setting and for both competitors, nodes in EU and US can reach a quorum with a low latency, and both of them suffer from the low performance of the Mumbai's node. Also, although they rely on different techniques to decide ordered commands, in this setting the CPU cycles needed to handle incoming messages are comparable.

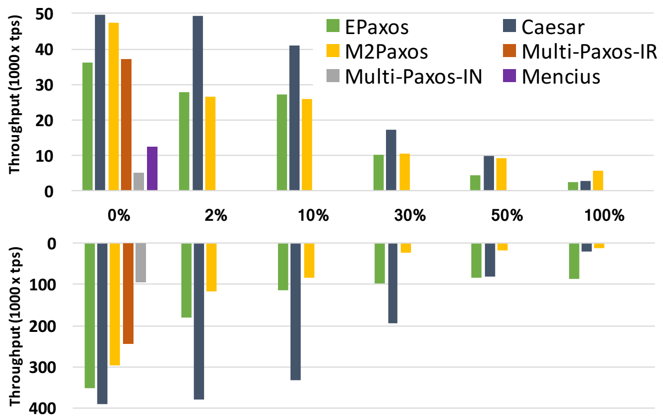


Fig. 9. Throughput by varying the percentage of commands from conflict pool. Batching is disabled in the top plot and enabled in the bottom plot.

In the bottom part of Figure 9, batching has been enabled. Mencius's implementation does not support batching thus we omitted it. The trend is similar to the one observed with batching disabled. The noticeable difference regards the performance of EPaxos when the percentage of conflicts increases. When 50% and 100% of commands are from the conflict pool, EPaxos behaves better than other competitors because, although the time needed for analyzing the conflict graph increases, it does not deploy a wait condition that contributes to slow down the ordering process if conflicts are excessive. In terms of improvements, CAESAR sustains its high throughput up until 10% commands are from the conflict pool ; it provides more than 320k ordered commands per second, which is almost 3 times faster than EPaxos. Multi-Paxos shows an expected behavior: it performs well under its optimal deployment, where the leader can reach consensus fast, but its performance degrades substantially if the leader moves to a faraway node.

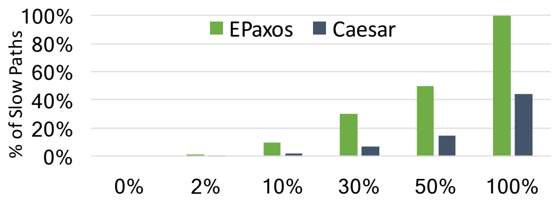


Fig. 10. % of commands delivered using a slow decision by varying % of commands from conflict pool. Batching here is disabled.

CAESAR's ability to take fewer slow decisions than existing consensus protocols in presence of conflicts helps it to achieve a lower latency and higher throughput than competitors. In Figure 10, we show the percentage of commands that were committed by taking fast decisions in both the protocols. It should be noted that the number of slow decisions taken by EPaxos is in the same range as the percentage of conflict. However, that is not the case of CAESAR, where the number of slow decisions more gracefully increases along with conflicts. In fact, CAESAR takes more than 3 times fewer slow decisions compared to EPaxos even under moderately conflicting (e.g. 30% commands from conflict pool) workloads. The reason for that is the wait condition that provides the rejection of a command only when its timestamp is invalid. In this experiment, to avoid confusion in analyzing statistics, batching has been disabled.

3.5.2 Recovery

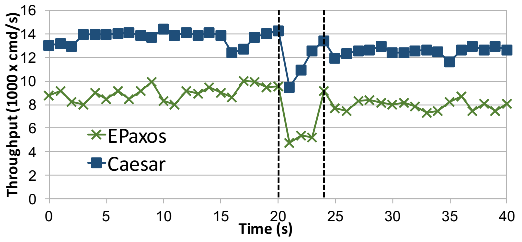


Fig. 11. Throughput when one node fails.

In Figure 11, we report the throughput when one node crashes, to show that it does not cause system's unavailability. We compared CAESAR and EPaxos. For this test, the requests are injected in a closed-loop with 500 clients on each node. After 20 seconds through the experiment, the instances of CAESAR and EPaxos are suddenly terminated in one of the nodes. Then, the clients from that node timeout and reconnect to other nodes. This is visible by observing the throughput falling down for few seconds due to loss of those 500 clients. However, as the clients reconnect to other available nodes and inject requests, the throughput restores back to the normal. In our experiment, the recovery period lasted about 4 seconds.

4 SPECTRUM

We now introduce SPECTRUM, a framework for switching consensus protocols online, starting from the motivations.

4.1 Motivations for a Consensus Framework

The need for a consensus framework to support consistent and performant coordination is due to the fact that in literature there exist a plethora of consensus solutions, each of which is optimized for some specific configurations of workload and deployment scenarios.

Over the past decade, multiple leaderless protocols have been introduced each providing different performance guarantees for various usecases. Most of these protocols implement the Generalized Consensus specification, where the total order is obtained only among non-commutative commands. Hence, the performance of such protocols have largely been dependent on the amount of contention level, which is measured in terms of percentage of conflicts among commands from different nodes, in the system. This observation enabled grouping protocols into families by their similarities in performance. Specifically, the contention level determined, in many cases, the family of protocols that stood out from others for providing the lowest values of latency and the highest values of throughput. This clearly motivated the design of SPECTRUM, a framework that facilitates transition between consensus protocols to achieve the highest possible performance.

Need for transparent transitions Although adaptivity in distributed systems is a well explored problem with many solutions, the design of an adaptive consensus framework for a combination of leaderless and leader-based protocols is unexplored because: *(i)* Despite consensus being a widely investigated topic, leaderless and more scalable implementations appeared in literature recently and are not well established yet; *(ii)* Reconciling protocols that adopt fast quorums with the ones that adopt classic quorums is challenging, especially if those protocols provide different guarantees in terms of safety and liveness properties, and carry different representations of the system state, e.g., dependency graph in EPaxos, ownership maps in M^2PAXOS, timestamp-based orders in Caesar, total order in Multi-Paxos.

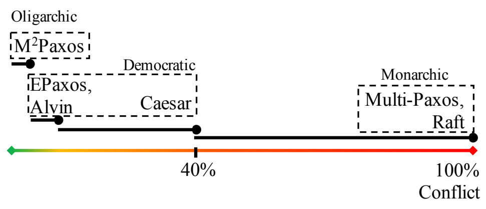


Fig. 12. Optimums for some consensus protocols in the conflict spectrum

A desirable adaptive consensus should switch from one specific consensus protocol to another one at runtime, independently of whether the protocols belong to the same family or not, in order to adapt the consensus layer to the characteristic of the workload and deployment scenarios. In doing so, it is crucial that the switching mechanism provides the same safety guarantees of the protocols that it manages and coordinates, while being transparent to the user. Moreover, the decision to switch from one protocol to another must be fault-tolerant; a decision to switch to a new protocol instance should persist even in the presence of crash faults. This means ruling out any solution that involves making unilateral decisions about the switch.

Note that this work focuses on reconfiguration among different consensus protocols, and not within a single protocol. For instance, this work does not provide a description of how to reconfigure the internal characteristics of Caesar (leaderless) to act like a single-leader one. Instead, SPECTRUM composes protocols as they are, and to do so, we leverage certain ideas from existing work on reconfiguration [22] and builds a solution using it. Moreover, leaderless protocols are newer, and reconfiguration among leaderless and leader-based protocols have not been explored to the best of our knowledge. This paper aims to address this gap. Moreover, we emphasize that leaderless protocols have their advantages and advocate their use in practice, but when they fail, SPECTRUM can alleviate the damage. SPECTRUM provides a protocol-agnostic switching scheme that can maximize performance under contention.

4.2 Description

SPECTRUM provides dynamic and transparent switches, fault-tolerant and non-blocking decisions, which enables adaptivity to different workload and deployment configurations, by adopting a novel switching solution that is based on consensus. In particular, the core idea is abstracting the whole switching and adaptation functionalities into a higher level consensus layer. Such a layer coordinates the switch between consensus protocols, while minimizing the unavoidable increase in user-perceived latency during transitions. Having a meta layer of consensus coordinating a switch from one consensus protocol to another, and taking care of the lifetime of the protocols, is a novel challenge.

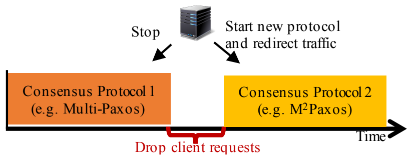


Fig. 13. Naive *stop-and-restart* protocol

Intuition. Before understanding the main intuition behind SPECTRUM, let us consider a simple but intuitive switching scheme in which some node triggers and coordinates a transition. The coordinator node stops the execution of one consensus protocol and starts the execution of a new protocol, like in Figure 13. We call this *stop-and-restart* solution. There are two main problems in such a setup: firstly, the coordination of the transition is not fault-tolerant, leaving the system vulnerable to crashes during the switching of a protocol; secondly, it is not clear whether commands that are submitted during the transition have to be rejected or not and, if not, which is the protocol in charge of deciding those commands. On the other hand, if commands are rejected, the system would suffers from a downtime.

To solve this, we leverage a result presented in [23], where the authors show that in state machine replication, the separation between agreement and execution is both necessary and sufficient to enable lazy recovery. Lazy recovery is a flexible solution to solve the problem of on-demand instantiation of replica nodes; the aim is to activate a minimal number of replicas first, and as they fail, activate the backup ones.

Indeed, the problem of on-demand instantiation in state machine replication is similar to the problem of instantiation of a new consensus protocol and the transfer of execution state. This leads to our solution, whose high level idea is depicted in Fig. 14. Solving consensus involves agreeing on the order of commands, and then executing them in that order. By recognizing that fault-tolerant transition among consensus protocols can actually be guaranteed via consensus itself, SPECTRUM adopts a solution where a meta consensus layer triggers and coordinates a switch, and uses the separation between agreement and execution components in both the pre-switch and post-switch consensus instances to guarantee a linear delivery of commands. The correctness of such a modularized approach has been also formally proved in [24].

The benefits of SPECTRUM are remarkable. Commands are never rejected during the transition period since the agreement layer provides continuity. However, they can experience an increase in latency due to a possible gap in time between the end of a pre-switch execution and the beginning of a post-switch execution. This is unavoidable since commands have to be delivered according to an order such that commands processed by the pre-switch agreement should be ordered before commands processed by the post-switch agreement. The next section describes SPECTRUM and its subsystems in detail with examples. The algorithmic details are deferred to [25], due to space constraints.

4.2.1 *Meta-Consensus*

To ensure full transparency to the user, it is crucial that SPECTRUM provides the same guarantees of the underlying consensus protocols. Therefore, an implementation of consensus itself as the core mechanism is the best fit for coordinating the protocol transitions in a non-blocking, fault-tolerant, and consistent way. This would also enable the modularization of the consensus protocols themselves, which can be integrated into a higher level consensus-based coordination module, which we call *Meta-Consensus*, as plugins.

Meta-Consensus module comprises of two components: the *agreement component* and the *management component*. The *agreement component* is the core consensus that decides on the next `Switch` command, which contains information on the switch itself (e.g., next consensus protocol), to be applied in a fault-tolerant manner. The *management component* is responsible for executing the decisions from the agreement component. Particularly, it manages the instances of the consensus protocols that are involved in the switch; it coordinates those instances in such a way that the commands that are submitted by the clients are correctly ordered, even if they have been submitted during the transition.

An example run of SPECTRUM is depicted in Figure 14. It shows all the steps that the system takes for commands that are submitted by clients before, during, and after a transition: the commands ▲, ■, and ◆, respectively. It also shows the execution of a transition itself. As described in the intuition part of Section 4.2, we divide the run of a consensus protocol in two parts, agreement and execution.

The figure depicts three nodes that participate in the agreement and execution of commands, and it is organized as follows. The first stripe represents the steps of the *agreement*

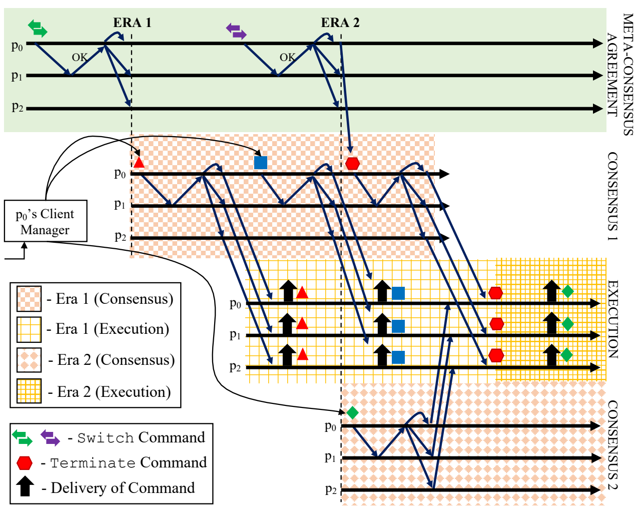


Fig. 14. An example execution of SPECTRUM. Every node has a client manager that manages client connections and directs client commands to the active consensus protocol.

component of *Meta-Consensus*; the second and the fourth ones represent the agreement parts of the instance of consensus before the switch, i.e., pre-switch consensus, and the instance of consensus after the switch, i.e., post-switch consensus, respectively; the third one represents the execution part of the pre-switch consensus followed by the execution part of the post-switch consensus. Note that the *management component* is not explicitly shown in the figure, but is discussed below.

Agreement Component. Most of the existing consensus protocols in literature — both single leader-based and leaderless — are viable candidates as implementations of the *agreement component*. Indeed, they have to implement agreement among nodes to decide the `Switch` command. The `Switch` command is a SPECTRUM command that encodes information necessary to perform the switch. It contains the identifier for the next consensus protocol that the *management component* must instantiate and redirect client traffic to. The result of execution of this command informs whether the switch was successful or not.

The *agreement component* implements a leader-based consensus algorithm for simplicity to decide on the `Switch` command. The leader informs all the nodes that the transition has to happen. In the figure, SPECTRUM takes steps for the `Switch` command twice. The first one is just to show that the switch can be used to initialize the first instance of a consensus protocol at the beginning of a run, while the second one is to show how to trigger an actual transition.

The agreement and execution of a command that has been submitted before a transition proceeds normally according to the active consensus instance. For example, command ▲ in the figure, undergoes an agreement phase first (second stripe) and then an execution phase (third stripe). For simplicity in the representation, we adopt a single-leader protocol – Multi-Paxos – as both pre-switch and post-switch consensus protocols. Therefore, the agreement of command ▲, as well as all the commands that are submitted by the clients, follows the algorithm of Multi-Paxos in Figure 14.

Moreover, it is sufficient to use a simpler protocols such as Multi-Paxos or Raft for the *agreement component*. *Meta-Consensus* only receives a command during a switch and is not susceptible to concurrent requests from nodes.

Management Component. Once the *agreement component* decides on the next `Switch` command, it sends the command to the *management component* for execution. The execution of `Switch` involves the creation of a new consensus protocol instance, as specified in the command, and redirection of client requests to the newly created instance. The *management component* also serves as the entry point for client commands into the client-facing consensus instances that it manages. Note that client commands do not pass through the *agreement component* of *Meta-Consensus*, but directly enter the management component. Only `Switch` commands pass through the agreement component.

When the *management component* receives the `Switch` from the *agreement component*, it creates the instance of the new consensus protocol as defined in the command, and initializes it. Thanks to the properties of consensus, the `Switch` is delivered to all the correct instances of the *management component* in the system, hence all the correct nodes creates and initializes the instance of the new consensus protocol that SPECTRUM is going to switch to. In Figure 14, when p_0 informs all that the switch should happen, nodes independently create a new instance of consensus in the fourth stripe, which can start processing command ◆.

Through the rest of the paper, we call the period during which a consensus protocol instance lives as an *era*. An *era* is started whenever a `Switch` is executed and a new instance of a protocol is created as a result, and hence it is strictly tied to a consensus instance. For example, from left to right in Figure 14, the first vertical line denotes the start of *era* 1, and the second one denotes the start of *era* 2. The core idea is that whenever a new *era* is started in a node, then all other nodes in the system will eventually start the new *era*, thanks to the properties of consensus that characterize the *agreement component*. Those nodes are the participants in the post-switch consensus instance.

In addition to the aforementioned tasks, the *management component* is also responsible for ensuring the in-order execution of the client commands with respect to different client-facing consensus protocols. Recall that a new *era* is created whenever a `Switch` is executed, and consequently, after that the ordering of new commands happens in the new *era*. This means that, at any given time during the switch, both the consensus instances are ordering commands that are submitted from the clients. The instance in the old *era* orders commands that were pending at the time the transition started, while the instance in the new *era* orders commands that are submitted from the start of the transition on. This way, the management component enables continuity of the service, with no downtime in ordering the commands. For example, in Figure 14, command ■, which was already in the system when the switch has started, is decided in *era* 1, while command ◆, which had been submitted by the client after the switch, is decided in *era* 2.

However, the execution happens only in one *era* at a time to establish a total order between commands in different *eras*. To facilitate this, the commands in the new *era* that are ready for execution are buffered until the previous *era* terminates. This means that command ◆ in the figure, which belongs to *era* 2, cannot be executed until commands ▲ and ■, which belong to *era* 1, have been. The *management component* takes care of that, and ensures that the commands are dispatched for execution first in the order that is decided by *Meta-Consensus* and then, within an *era*, by the order that is decided by the respective instance of consensus that is running in that *era*. The way SPECTRUM defines the completion of an *era* depends on the characteristics of the consensus protocols that are involved in the switch, and it is described in Section 4.2.2.

4.2.2 Switching between Consensus Protocols

Meta-Consensus instantiates a new *era* and switches the traffic to that *era*. The core problem of the switching mechanism is the identification of completion of a previous *era*, since the *management component* of *Meta-Consensus* should ensure that commands across *era*s are executed sequentially. To enable the *management component* to learn when commands in a previous *era* have been executed, we adopt a technique used in Stoppable Paxos [26]. The idea is to propose a special `Terminate` command to the previous *era* protocol that acts as a marker indicating the end of commands in that *era*. Once `Terminate` command is delivered to the management component, it can start executing commands in the new *era*.

Despite the differences in the nature of existing families of consensus (see Section 4.1), SPECTRUM proposes a uniform method for handling the delivery of `Terminate` commands. For clarity, we describe two mechanisms – one for transition from single leader-based and another from leaderless protocol – and later combine them into one, for simplicity.

Transition from Single Leader-based Consensus. For single-leader (or *monarchic*) protocols (Multi-Paxos and Raft), it is sufficient to send the `Terminate` command to a leader node, and let that node establish the termination of an instance in an *era*. In Figure 14, p_0 first requests a quorum of nodes to accept the termination of the consensus instance at *era* 1, and then forces them to terminate that instance. Meanwhile, new client commands are ordered in the new *era*, but are buffered for execution until the old *era* terminates.

When the `Terminate` command is delivered to the *management component* at a node, it transitions to execute commands in the new *era*, e.g., command ◆ in the figure. The `Terminate` command ensures the total order of the commands between two consecutive *eras*. The properties of consensus ensure that commands ▲ and ■ belonging to Consensus 1 in the figure have already been processed and executed by a node at the time the final decision for `Terminate` is processed by that node.

It should be noted that the decision on the termination is fault-tolerant even though only one node takes care of the decision. This is because that node is anyway the leader of a consensus instance and, as such, it must ensure that the decision is stable before forcing it to all the others.

Transition from Leaderless Consensus. Leaderless (e.g., *democratic*, *oligarchic*) protocols typically implement the Generalized Consensus specification, in that the total order is obtained only among non-commutative commands. In addition, every node can propose and decide on the outcome of their commands. This entails that the `Terminate` command must find a total order among all the commands submitted to the system by all nodes.

In general, the commands submitted to Generalized Consensus protocols must embed some commutativity information. In practice, this is achieved by using the identities of the application objects that the commands operate on. Thus, by including the identities of all application objects, the `Terminate` command can establish a total order among all the commands submitted to the system. One optimization is to include the range of identities rather than every single identity in the command, which practical systems support [4].

4.3 Correctness Arguments

We show how SPECTRUM guarantees the properties of consensus, as described in Section 2. Due to space constraints we do not provide a complete formal proof. However, we provide the reader with a very intuitive explanation on the correctness of SPECTRUM, which partially relies on the correctness of the underlying consensus protocols.

SPECTRUM guarantees *Non-triviality* by construction of the algorithm, which decides at least the commands that are proposed, and by the assumption on the network layer, which neither creates nor duplicates messages. *Stability* is trivially guaranteed since commands are decided one-by-one, and the decisions are never retreated. Furthermore, a proposed command will be eventually decided since SPECTRUM proposes it to one of the underlying consensus protocol instances that it is coordinating, and therefore it provides the same liveness guarantees of those protocols.

However, there are two exceptions. – *i)* the command has been proposed to an instance of consensus that might not guarantee liveness due to the amount of contention, e.g., M^2PAXOS, and hence is never learnt – *ii)* the command is learnt by an instance of consensus after the instance decides a `Terminate` command, which means that the command cannot be decided, at least in the era associated with that instance. In both cases, the command remains pending, and the client that submitted it never receives a reply for it. The solution relies on command re-transmission: a client that does not receive a reply for a command in a predefined amount of time, it can resubmits the command, which is eventually decided (if had not been decided before) by an underlying consensus instance after a possible switch. It is worth noticing that SPECTRUM eventually performs a switch in case a liveness problem arises due to contention.

Liveness is guaranteed during crashes as well. The critical scenario is when the leader that coordinates a switch crashes while it is broadcasting a CHANGEERA message to trigger a `Switch`. If that happens, then there exists at least one correct node that knows the information about the switch, since the leader has received a quorum of ACKACCEPT messages that are tagged as *ACK* before sending CHANGEERA. Furthermore, no more than $min\,\{\mathcal{CQ}\} - 1$ nodes can crashes, where $min\,\{\mathcal{CQ}\}$ is the minimum size of a quorum.

SPECTRUM guarantees *Consistency* for commands that are not submitted during a transition because the underlying consensus protocol instances guarantee *Consistency*. On the other hand, SPECTRUM guarantees that, if a command is learnt before a `Terminate` command on a node in *era* x, then all correct nodes learn that command before `Terminate` in x, thanks to the *Consistency* of the active consensus instance in x. Furthermore, if a command has been learnt after `Terminate` had been decided in *era* x, then the command is either not decided or it has been learnt in *era* $y > x$. Since commands that are learnt by a node in *era* y cannot be decided by the *execution* instance of y until the commands that are lerant by that node in every *era* $x < y$ have been decided by the execution instance of x, *Consistency* follows.

4.4 Experimental Evaluation

We prototyped SPECTRUM in Java and incorporated the following consensus protocols into the framework: Multi-Paxos, CAESAR, and M^2PAXOS. Each of these protocols are fundamentally diverse, and they effectively demonstrate the capabilities of SPECTRUM in carrying out the switch. Moreover, these three protocols cover the majority of the conflict spectrum (Fig. 12) and allow us to demonstrate SPECTRUM's ability to cope with all contentious workloads. To demonstrate the effectiveness of our approach, we designed three different experiments, each highlighting a unique aspect of the framework. The specifics of the experiments are discussed in the following subsections.

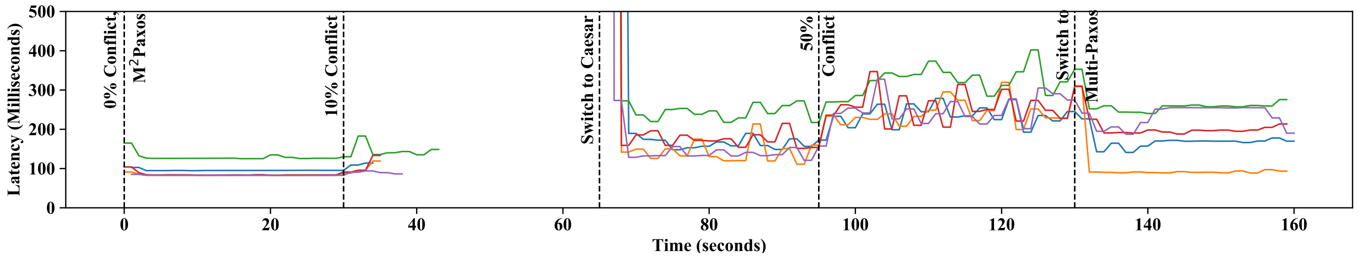
The deployment configuration is identical to that in Section 3.5. The experiments measure latency at different sites by placing 50 clients/site (250 in total) and injecting requests in a closed loop.

4.4.1 Minimizing latency under conflicts

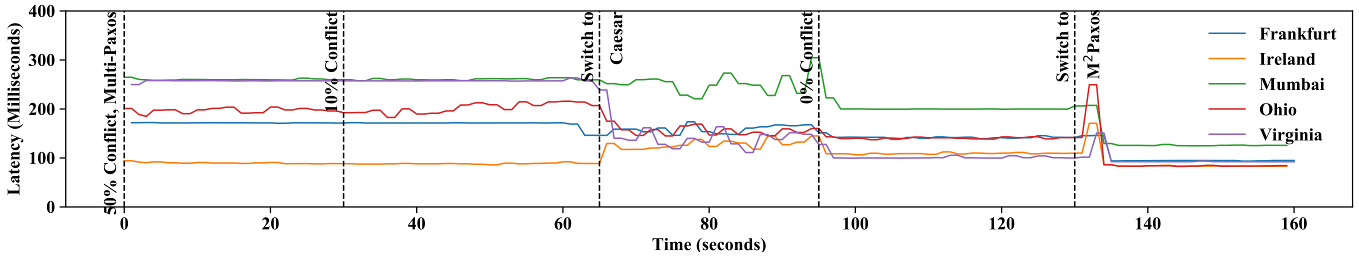
In this experiment, we show that SPECTRUM's ability to switch consensus protocols at runtime enables to provide the minimum possible user-perceived latency for any contentious workload. In other words, as the contention level in the workload changes, SPECTRUM can switch to a different protocol to reduce the user-perceived latency. Contention level is adjusted by changing the amount of concurrent conflicting commands entering the system. For example, at 10% contention, every client in the system issues a non-commuting command once every ten commands (at random) that conflicts with another such concurrent command. To show this, we devised a static oracle that observes the amount of conflict in the workload injected to the system and triggers the switch to a protocol that can better handle that particular workload. The condition for the oracle to trigger the switch depends on the percentage of conflicts: for ranges of $[0, 10)$, the oracle chooses M^2PAXOS; between $[10, 50)$ the oracle chooses CAESAR; else the oracle chooses Multi-Paxos.

We conducted two sub-experiments using the oracle. In the first experiment, we increase the contention level over time in order to provoke a switch. The result is presented in Figure 15(a). The x-axis presents the time in seconds since the beginning of the experiment, and the y-axis shows user-perceived latency in milliseconds.

We initialize SPECTRUM with M^2PAXOS as the starting consensus protocol and inject a non-conflicting workload. In this case, the oracle does not trigger any switch as M^2PAXOS is the best protocol for this workload. At time $t = 30s$, the amount of conflicts in the workload is increased to 10%. At this point, M^2PAXOS experiences a livelock caused by conflicting ownership acquisitions, and thus the client requests timeout. The oracle steps in at $t = 65s$ and switches to CAESAR, and in few seconds, SPECTRUM responds and delivers commands to the client. At $t = 95s$, we increase the conflict to 50%, and CAESAR starts performing poorly, but not as worse as M^2PAXOS at 10% conflict. The oracle triggers



(a) Scenario showing the effectiveness of the switch as the contention in the workload increases over time.



(b) Scenario showing the effectiveness of the switch as the contention in the workload decreases over time.

Fig. 15. Effectiveness of SPECTRUM in providing minimum possible latency for any conflicting workload.

the switch to Multi-Paxos at $t = 130s$, and this reduces the latency as at this amount of conflict, a monarchic protocol is better than any other group of protocols.

In the second experiment, we reversed the process; that is, we reduced the conflicting percentage from 50% to 0% over a time period. The result is in Figure 15(b). Here, the system is initialized with Multi-Paxos, and a 50% conflicting workload is injected initially. The oracle does not trigger any switch as conditions are suitable for Multi-Paxos. At $t = 30s$, the conflict drops from 50% to 10%, making CAESAR more preferable. Therefore, at $t = 65s$, the oracle triggers a switch to CAESAR, and the latency drops down immediately. At $t = 95s$, we reduce the conflict to 0%, and this workload can be better served by M^2PAXOS. Therefore, such a switch is triggered at $t = 130s$ by the oracle and latency drops again.

Note that we configured our oracle to perform the switch after 35 seconds of observing a new workload and keep the conflict rate same for about 60 seconds. This is done to explicitly contrast the change in latency due to the new workload before and after the switch.

4.4.2 Comparison against an alternate switching scheme

We implemented *Stop-and-restart*, an alternate switching scheme that follows the one shown in Figure 13, to contrast the merits of our solution, during a transition. In *Stop-and-restart*, an external coordinator implements the switch by first stopping the current instance of the consensus protocol, ensuring that all pending commands have been decided and executed, and then starting the new instance of a different consensus protocol. Note that, in contrast to SPECTRUM, *Stop-and-restart* rejects any new commands that is submitted to the system during the transition, and is not fault-tolerant.

In Figure 16, we contrast the performance of SPECTRUM and *Stop-and-restart* during the switch from Multi-Paxos to M^2PAXOS. The workload has been kept constant with 0% conflict throughout the experiment to highlight the cost of the `Switch`. It can be observed that the switch in SPECTRUM is transparent to the client, while in *Stop-and-restart*, the presence of an external coordinator to handle

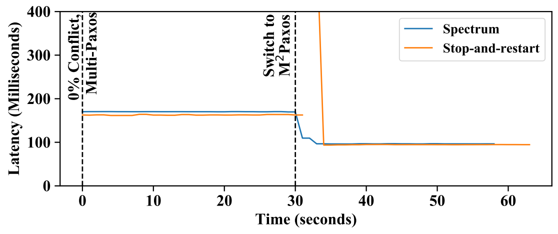


Fig. 16. SPECTRUM vs *Stop-and-restart*. The former performs the switch oblivious to the client, while the latter drops requests during the switch.

the switch and a non-streamlined transition causes the client requests to timeout, and once the switch completes, client requests receive their responses. This is shown in the figure with a period of no latency reporting (indicating the transition period) followed by a sharp spike (indicating switch completion and client response).

4.4.3 Fault Tolerance

We devised a specific experiment to show the fault-tolerance of SPECTRUM. We instrumented *Meta-Consensus* to force a crash of the leader during the agreement on the switch. Specifically, right before broadcasting DECIDE. As a result, the failure detector of a correct node performs the recovery to finalize the decision of the pending `Switch` command.

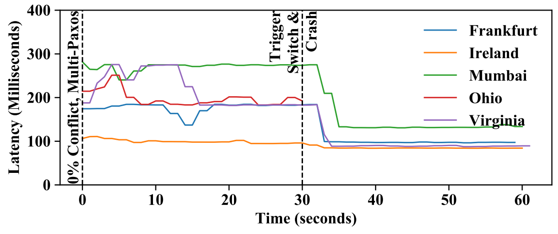


Fig. 17. Fault Tolerance of SPECTRUM.

The result is shown in Figure 17. At $t = 30$, the leader in Ohio is forcefully crashed. Due to the crash, only the clients of that node timeout, while clients of other nodes keep receiving responses from the active consensus protocol. Other clients do not observe any difference in latency since the only node that benefitted from Ohio was Virginia, e.g., having Ohio is its quorums. However, Virginia can access Frankfurt with almost the same latency as Ireland, which already was in the quorums.

5 Related Work

Consensus Solutions In the Paxos [3] algorithm, a value is decided after a minimum of four communication delays. Progress guarantees cannot be provided as the initial prepare phase may fail in the presence of multiple concurrent proposals. Multi-Paxos alleviates this by letting promises in the prepare phase cover an entire sequence of values. This effectively establishes a distinguished proposer that acts as a single designated leader.

Fast Paxos [27] eliminates one communication delay by having proposers broadcast their requests and bypass the leader. However, a classic Paxos round executed by the leader is needed to resolve a collision, reaching a total of six communication delays to decide a value. Generalized Paxos [13] relies on a single leader to detect conflicts among commands and enforce an order, and it uses fast quorums as Fast Paxos. On the contrary, CAESAR avoids the usage of leader either to reach an agreement, as in Paxos, or to resolve a conflict, unlike in Fast and Generalized Paxos.

Mencius [11] overcomes the limitations of a single leader protocol by providing a multi-leader ordering scheme based on a pre-assignment of slots to nodes. It pre-assigns sending slots to nodes, and a sender can decide the order of a message at a certain slot s only after hearing from all nodes about the status of slots that precede s. Clearly this approach is not able to adopt quorums (unlike Paxos), and it may result in poor performance in case of slow nodes or unbalanced inter-node delays. To alleviate the problem of slow nodes, Fast Mencius [28] uses a mechanism that enables the fast nodes to revoke the slots assigned to the slow nodes. However, Fast Mencius still suffers from high latency in specific WAN deployments since it does not rely on quorums for delivering.

EPaxos employs dependency tracking and fast quorums to deliver non-conflicting commands using a fast path. In addition, its graph-based dependency linearization mechanism that is adopted to define the final order of execution of commands may easily suffer from complex dependency patterns. Instead, Alvin [12] avoids the expensive computation on the dependency graphs enforced by EPaxos via a slot-centric decision, but it still suffers from the same vulnerability to conflicts as EPaxos: a command's leader is not able to decide on a fast path if it observes discordant opinions from a quorum of nodes. In contrast, CAESAR's fast decision scheme is optimized to increase the probability of deciding in two communication delays regardless of discordant feedbacks.

M^2Paxos [14] is a multi-leader consensus implementation that provides fast decisions while *i)* adopting only a simple majority quorum, and *ii)* avoiding the exchange of command dependencies. It does that by embedding an ownership acquisition phase for commands into the agreement process, so as to guarantee that a node having the ownership on a set of commands can autonomously take decisions on those commands. However, in case there are multiple nodes that compete for the decision of non-commutative commands, the protocol might require an expensive ownership acquisition phase to re-distribute their ownership records.

CAESAR is related to Clock-RSM [29]. In Clock-RSM, each node proposes commands attached with its physical timestamp, which are then deterministically ordered according to their associated timestamps. Although Clock-RSM is multi-leader like CAESAR, and it relies on quorums to implement replication, it suffers from the same drawbacks of Mencius, namely the need of a confirmation that no other command with an earlier timestamp has been concurrently proposed.

Composition and Adaptation. Composition of consensus protocols has been investigated in [30]. `Abstract` is an abstraction for designing and reconfiguring generalized state machines, by leveraging the idea of composing instances of different fault-tolerant consensus protocols. The idea is to build simpler consensus protocols each tolerating particular system conditions such as fault models and contention, and compose them together to achieve a robust system. The downside of the `Abstract` approach is that it requires the candidate protocols to implement specific interfaces. Specifically, the candidates must be able to export (import, respectively) internal state outside (into, respectively) the protocol. This means that existing as well as new protocols must be rethought to accommodate to the abstraction.

SPECTRUM, in contrast, provides isolation and composition without requiring any changes to candidate consensus protocols. Specifically, the protocols under consideration maintain state differently and thus transfer of such state from one protocol to another is a tedious, perhaps an impossible task. SPECTRUM treats candidate protocols as black-boxes and builds a general solution such that no state transfer is necessary. This makes our framework more appealing for existing as well as new consensus designs.

In [22], the authors describe algorithms for reconfiguring state machines, including the one presented in [26]. The authors propose using a special `Terminate` command whose delivery will stop the active protocol, after which the reconfiguration such as number of nodes and failure tolerance can be adjusted. This is similar to the *stop-and-restart* solution, with which we compared our approach. Moreover, as mentioned in Section 4.2.2, SPECTRUM also uses the idea of using the `Terminate` command from [26]. However, SPECTRUM stands out distinctly due to its ability to perform the switch seamlessly while ensuring fault-tolerance. In addition, we present a complete algorithm that has been implemented and experimentally evaluated, unlike [22], [26].

Adaptivity in distributed systems: Our solution is related to self-tunable solutions that adapt their internal mechanisms to react to changes in the environment. SPECTRUM specifically focuses on the mechanisms offering an effective and cheap way for the adaptation than on the oracles that trigger the adaptation. The design of the oracles is an orthogonal problem, and solutions for that can be used as plugins to the framework. TAS [31] is an approach for automating the elastic scaling of replicated in-memory transactional systems. SPECTRUM can benefit from its performance predictor that relies on the combined usage of analytical modeling and machine learning, since it is able to forecast the effects of data contention. For the same reason, the machine learning-based model of MorphR [32] can be exploited by SPECTRUM, which finds the optimal transactional replication protocols according to the conflicts in the system. MorphR is able to choose between blocking, i.e., 2-PC, and non-blocking, i.e., total order, protocols, but it does not focus on the optimal switching mechanisms among non-blocking protocols, e.g., different consensus protocols.

6 Conclusion

This paper presents two contributions to tame contention in consensus-based distributed systems. CAESAR addresses the performance degradation in existing systems under low-to-moderate contention, while SPECTRUM provides a framework for switching consensus protocols online to adapt to changing workload conditions and provide maximal performance at any workload with an apt consensus protocol.

7 Acknowledgments

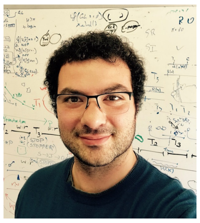
A preliminary version of this work appeared in [15]. This work is supported in part by US National Science Foundation under grant CNS 1523558 and AFOSR under grants FA9550-15-1-0098 and FA9550-16-1-0371. The work was done while the authors (S. Peluso, R. Pelmieri and G. Losa) were at Virginia Tech.

References

[1] B. Charron-Bost and A. Schiper, "Uniform Consensus is Harder Than Consensus," *J. Algorithms*, vol. 51, no. 1, pp. 15–37, Apr. 2004.
[2] L. Lamport, "The Part-time Parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.
[3] ——, "Paxos made simple," *ACM Sigact News*, 2001.
[4] J. C. Corbett *et al.*, "Spanner: Google's globally-distributed database," in *10th USENIX OSDI*, 2012, pp. 261–264.
[5] S. Hirve, R. Palmieri, and B. Ravindran, "Archie: A Speculative Replicated Transactional System," in *Proceedings of the 15th International Middleware Conference*, ser. Middleware, 2014, pp. 265–276.
[6] T. Kraska *et al.*, "MDCC: Multi-data Center Consistency," in *EuroSys*, 2013, pp. 113–126.
[7] H. Mahmoud *et al.*, "Low-latency Multi-datacenter Databases Using Replicated Commit," *Proc. VLDB Endow.*, vol. 6, no. 9, pp. 661–672, Jul. 2013.
[8] J. Gray and L. Lamport, "Consensus on Transaction Commit," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 133–160, Mar. 2006.
[9] K. Bogdanov *et al.*, "The nearest replica can be farther than you think," in *Proceedings of the Sixth ACM SoCC*, 2015, pp. 16–29.
[10] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is More Consensus in Egalitarian Parliaments," in *SOSP*, 2013, pp. 358–372.
[11] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building Efficient Replicated State Machines for WANs," in *OSDI*, 2008.
[12] A. Turcu *et al.*, "Be General and Don't Give Up Consistency in Geo-Replicated Transactional Systems," in *OPODIS*, 2014.
[13] L. Lamport, "Generalized Consensus and Paxos," Microsoft Research, Tech. Rep. MSR-TR-2005-33, March 2005.
[14] S. Peluso *et al.*, "Making fast consensus generally faster," in *DSN*, 2016, pp. 156–167.
[15] B. Arun *et al.*, "Speeding up Consensus by Chasing Fast Decisions," in *IEEE/IFIP DSN*, 2017.
[16] 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, Apr. 1985.
[17] R. Guerraoui and A. Schiper, "Genuine Atomic Multicast in Asynchronous Distributed Systems," *Theor. Comput. Sci.*, vol. 254, no. 1-2, pp. 297–316, Mar. 2001.
[18] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Springer, 2006.
[19] L. Lamport, "Future directions in distributed computing," A. Schiper *et al.*, Eds., 2003, ch. Lower Bounds for Asynchronous Consensus, pp. 22–23.
[20] B. Arun *et al.*, "Speeding up Consensus by Chasing Fast Decisions," Tech. Rep., 2017. [Online]. Available: https://arxiv.org/abs/1704.03319
[21] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.
[22] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *ACM SIGACT News*, vol. 41, no. 1, pp. 63–73, 2010.
[23] R. Shi and Y. Wang, "Cheap and Available State Machine Replication," in *USENIX ATC*, 2016.
[24] R. Guerraoui, V. Kuncak, and G. Losa, "Speculative Linearizability," in *Proceedings of the 33rd ACM PLDI*, 2012.
[25] B. Arun, S. Peluso, and B. Ravindran, "Spectrum: A Framework for Adapting Consensus Protocols," Tech. Rep., 2019. [Online]. Available: https://arxiv.org/abs/1902.05873
[26] L. Lamport, D. Malkhi, and L. Zhou, "Stoppable Paxos," Microsoft Research, Tech. Rep. MSR-TR-2008-192, April 2008.
[27] L. Lamport, "Fast Paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.
[28] W. Wei *et al.*, "Fast mencius: Mencius with low commit latency," in *IEEE INFOCOM*, 2013, pp. 881–889.
[29] J. Du *et al.*, "Clock-RSM: Low-Latency Inter-datacenter State Machine Replication Using Loosely Synchronized Physical Clocks," ser. DSN, 2014, pp. 343–354.
[30] R. Guerraoui *et al.*, "The next 700 bft protocols," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 363–376.
[31] D. Didona *et al.*, "Transactional Auto Scaler: Elastic Scaling of Replicated In-Memory Transactional Data Grids," *ACM Trans. Auton. Adapt. Syst.*, 2014.
[32] M. Couceiro *et al.*, "Chasing the Optimum in Replicated In-memory Transactional Platforms via Protocol Adaptation," in *IEEE/IFIP DSN*, 2013.

Balaji Arun received the BS and MS degrees in computer engineering from Virginia Tech, Blacksburg, VA. He is currently working towards the PhD degree in the Systems Software Research Group, Virginia Tech, led by professor Binoy Ravindran. His research interests include fault-tolerance and verification of distributed systems.

Sebastiano Peluso is a Performance Engineer at Facebook. Previously, he was a Research Assistant Professor at the Virginia Polytechnic Institute and State University. He received his MS in Computer Engineering in 2010 from Sapienza University of Rome, and his PhD in Computer Engineering in 2014 from Sapienza University of Rome and the Instituto Superior Técnico, Universidade de Lisboa. His research interests lie in the area of distributed systems and parallel programming, with focus on scalability and fault tolerance of transactional systems.



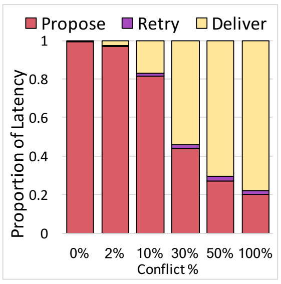
Roberto Palmieri is an assistant professor with the Computer Science and Engineering Department, Lehigh University where he leads the Scalable Systems Software Research Group. His research interest include designing and developing concurrency control protocols and mechanisms for multicore architectures, cluster and geographically distributed systems, with high programmability, performance, scalability, and dependability properties. He has received the 2017 AFOSR YIP and the Best Paper Award at ACM SYSTOR 2018.

Giuliano Losa received the PhD degree in computer science from EPFL, Switzerland. He was a postdoctoral researcher at Virginia Tech and UCLA. His research interests include distributed computing, distributed systems, and formal methods. He is currently employed at Galois, Inc., Portland, Oregon, where he works on making formal methods more accessible to software engineers building distributed systems.

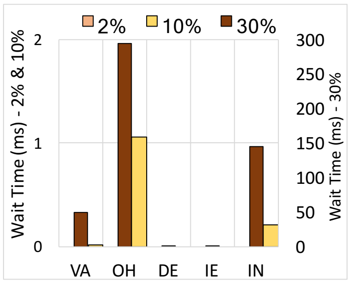
Binoy Ravindran is a professor of electrical and computer engineering at Virginia Tech, where he leads the Systems Software Research Group which conducts research on distributed systems, operating systems, virtualization, compilers, concurrency, and verification. His group has published more than 290 papers in these spaces, including eight best paper awards and nominations. Several of his group's results have been transitioned to the US DOD, in particular, the Navy. He has mentored six research faculty members, 14 postdoctoral scholars, and 18 PhD students, ten of whom currently hold tenured or tenure-track faculty positions. He is an ACM distinguished scientist, a former office of naval research faculty fellow, and serves or has served on the editorial boards of IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, ACM Transactions on Embedded Computing Systems, IEEE Design & Test, and IEEE Transactions on Sustainable Computing.

Appendix A
Additional Evaluation

A.1 Internal Statistics



(a) Ordering phases.



(b) Wait condition.

Fig. 18. Latency breakdown for CAESAR.

In Figure 18, we report the internal statistics of CAESAR gathered during the experiment in Figure 9 (see Section 3.5). Figure 18(a) shows the breakdown of the proportion of latency consumed by each ordering phase of the protocol. For no conflicts (e.g. 0%–2% commands from conflict pool), the maximum time is spent in the proposal phase. The cost of the delivery is very low, since there are no dependencies. However, as the percentage of commands from conflict pool increases, delivery becomes a major portion of the total cost because a STABLE command must wait for the delivery of all the conflicting commands with an earlier timestamp before being delivered. Figure 18(b) reports the average time spent on the wait condition during the proposal phase by conflicting commands using the same workload for throughput measurement. Note that we used a different scale (right y-axis) for 30% plot to highlight the difference with respect to the case of 2% and 10% plot. Close together nodes experience a quicker timestamp advancement than faraway nodes because they are able to exchange proposals faster. Faraway nodes are not aware of this advancement, thus they propose commands with a lower timestamp, which causes their conflicting commands to wait.

A.2 Latency in Different Deployment Configuration

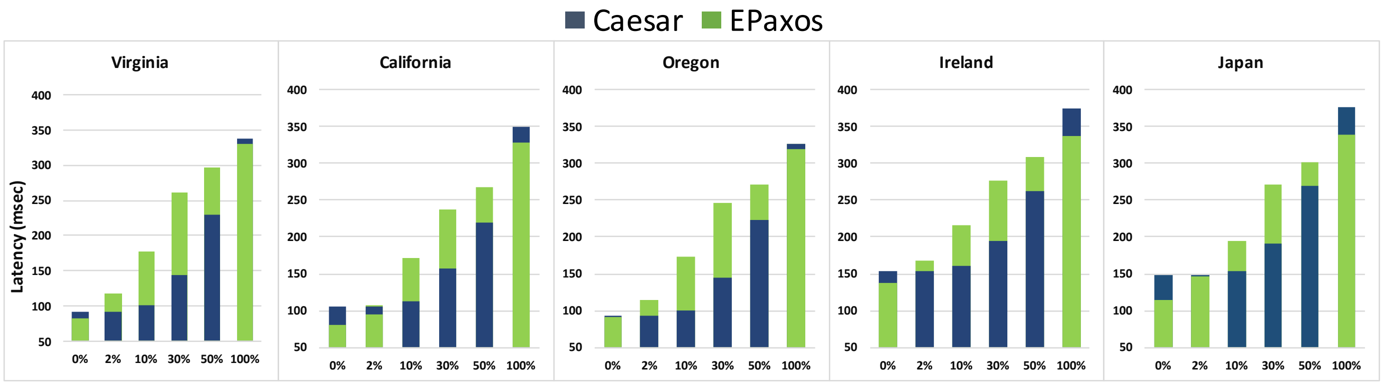


Fig. 19. Average latency for ordering and processing commands by changing the percentage of commands picked from conflict pool. Batching is disabled. Bars are overlapped

We, now, present details of an experimental evaluation with a different deployment setup and conflict categorization. The reason for this is two folds. First, the choice of deployment places a majority of replicas (3 out of 5) within a same continent in order to understand the performance of CAESAR when replicas are close together. Second, for conflicting commands, the key is picked from a pool of 50 keys rather than a pool of 100 keys. This increases the contention further than that in Section 3.5, particularly in a closed-loop latency experiment. We did not reduce the key space further to emulate realistic workloads. Note that the use of a conflict pool should favor EPaxos more than Caesar, since EPaxos does not have a wait condition. However, as we show below, the additional geographical RTT is more detrimental than the wait condition.

We deployed the competitors on five nodes located in Virginia (US), Oregon (US), Califronia (US), Ireland (EU), and Japan (Asia). It is worth noting that EPaxos' US nodes have a majority within the US itself due to its use of slightly smaller fast quorums, while Caesar should access a node outside of the US regardless. The round trip time (RTT) that we measured in between nodes within the US are all below 100ms, with the RTT between CA and OR at 20ms. The node in Japan experiences the following delays with respect to the other nodes: 147ms/Virginia, 104ms/California, 92ms/Oregon, 234ms/Ireland.

In Figure 19, we report the average latency incurred by CAESAR and EPaxos to order and execute a command. Given the latency of a command is affected by the position of the leader that proposes the command itself, we show the results collected in each site. Each cluster of data shows the behavior of a system while increasing the percentage of conflicts in the range of {0% – no conflict, 2%, 10%, 30%, 50%, 100%}.

Under no conflicts, EPaxos is able to provide lower latency by taking the fast path for all requests. As conflict increases, the latency increases because the commands are forced to take the slow path. However, that is not the case with CAESAR. It is able to maintain a lower latency compared to EPaxos by mostly staying on the fast path until up to 50% commands are from conflict pool. We can observe that CAESAR, on average, incurs up to 35% lower latency across all nodes than EPaxos when 30% of commands are from the conflict pool. However, when 100% of commands are from the conflict pool, CAESAR performs slightly slower than EPaxos, because the impact of the wait condition on performance is higher due to the fact that, at this conflict rate, the commands are almost totally ordered.