Speeding up Consensus by Chasing Fast Decisions

Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, Binoy Ravindran
ECE, Virginia Tech, USA
{balajia,peluso,robertop,giuliano.losa,binoy}@vt.edu

***Abstract*—This paper proposes CAESAR, a novel multi-leader Generalized Consensus protocol for geographically replicated sites. The main goal of CAESAR is to overcome one of the major limitations of existing approaches, which is the significant performance degradation when application workload produces conflicting requests. CAESAR does that by changing the way a fast decision is taken: its ordering protocol does not reject a fast decision for a client request if a quorum of nodes reply with different dependency sets for that request. The effectiveness of CAESAR is demonstrated through an evaluation study performed on Amazon's EC2 infrastructure using 5 geo-replicated sites. CAESAR outperforms other multi-leader (e.g., EPaxos) competitors by as much as 1.7x in the presence of 30% conflicting requests, and single-leader (e.g., Multi-Paxos) by up to 3.5x.**

***Keywords*-Consensus, Geo-Replication, Paxos**

I. INTRODUCTION

Geographically replicated (geo-scale) services, namely those where actors are spread across geographic locations and operate on the same shared database, can be implemented in an easy 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 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 services [4], [5], [6], [7], [8]. An example of Paxos used in a production system is Google Spanner [4], [9].

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

To overcome this limitation, protocols aimed at allowing multiple nodes to operate as command leaders at the same time [10], [11], [12] have been proposed. Such solutions provide implementations of Generalized Consensus [13], a variant of Consensus that agrees on a common order of non-commutative (or conflicting) commands. These approaches, despite avoiding the bottleneck of the single leader, suffer from other 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.

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 aspect: 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.

To address these aspects, we propose CAESAR, a consensus layer that deploys an innovative multi-leader ordering scheme. As a high-level intuition, when a conflicting command is proposed, CAESAR does not suffer from the condition that causes a slow decision of that command in all existing Generalized Consensus implementations (including EPaxos). Such a condition is the following:

For a proposed command c, at least two nodes in a quorum are aware of different sets of commands conflicting with c.

CAESAR avoids this pitfall because it approaches the problem of establishing agreement from a different perspective. When a command c is proposed, CAESAR seeks an agreement on a common delivery timestamp for c rather than on its set of conflicting commands. To facilitate this, a local *wait condition* is deployed to prevent commands conflicting with c from interfering with the decision process of c if they have a timestamp greater than c's timestamp.

The basic idea behind the ordering process of CAESAR is the following: a command is associated with a logical timestamp by the sender, and if a quorum of nodes confirms that the timestamp is still valid, then 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. Note that the equality of the sets of conflicting commands collected by nodes does not influence the ordering decision. With this scheme, CAESAR boosts timestamp-based ordering protocols, such as Mencius [11], by exploiting quorums, which is a fundamental requirement in geo-scale where contacting all nodes is not feasible. CAESAR does that without relying on a

single designated leader unlike Multi-Paxos.

Our approach also provides the benefit of a more parallel delivery of ordered commands when compared to EPaxos, which requires 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.

We conducted an evaluation study for CAESAR using key-value store interfaces. With them, we can inject different workloads by varying the percentage of conflicting commands and measure various performance parameters. We contrasted CAESAR against: EPaxos and M^2Paxos, multi-leader quorum-based Generalized Consensus implementations; Mencius, a multi-leader timestamp-based Consensus implementation that does not rely on quorums; and Multi-Paxos, a single-leader Consensus implementation. As a testbed, we deployed 5 geo-replicated sites using the Amazon EC2 infrastructure.

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 up to 60% and increasing throughput by 1.7×. These performance boosts are due to the higher percentage of fast decisions accomplished. With 30% of conflicting workload, CAESAR takes up to 70% fewer slow decisions compared to EPaxos.

II. Related Work

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 [15] eliminates one communication delay by having proposers broadcast their request 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. Some of its limitations are overcome by FGGC [16], which can use optimal quorum size but still relies on designated leaders. On the contrary, CAESAR avoids the usage of a single designated leader either to reach an agreement, as in Paxos, or to resolve a conflict, as 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 Consensus instances 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 has been proposed [17]. It 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 of EPaxos: a command's leader is not able to decide on a fast path if it observes discordant opinions from a quorum of nodes. That is not the case of CAESAR, whose fast decision scheme is optimized to increase the probability of deciding in two communication delays regardless of discordant feedbacks.

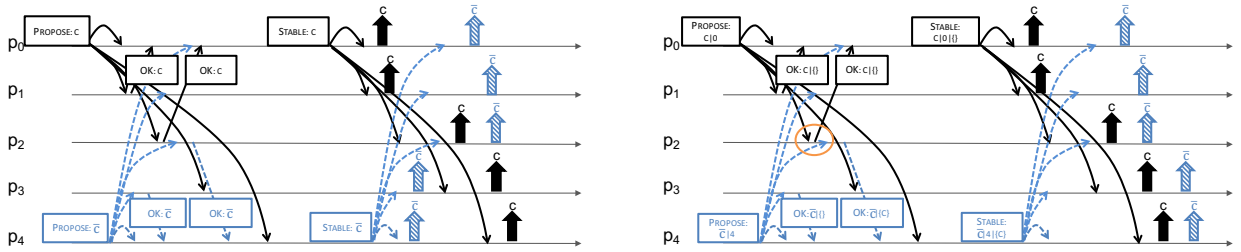
M^2Paxos [14] is a multi-leader consensus implementation that provides fast decisions while *i)* adopting only a majority of nodes as quorum size, and *ii)* avoiding to exchange dependencies of commands. It does that by embedding an ownership acquisition phase for commands into the agreement process, 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 also related to Clock-RSM [18]. In Clock-RSM, each node proposes commands piggybacked 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.

III. 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.

Because of FLP [19], we assume that the system can be enhanced with the weakest type of unreliable failure detector [20] that is necessary to implement a leader election service [21]. 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



(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 "only" via reliable 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. Reliable broadcast execution vs. CAESAR execution

with size at least equal to $\lceil \frac{3N}{4} \rceil$ (derived by minimizing $\mathcal{CQ}$). As it will be clear in Section V, a fast quorum is required to achieve fast decisions in two communication delays, while classic quorum is required in case 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 $C\text{-}struct\ cs$ via the DECIDE(cs) interface. The specification is such that: commands that are included in decided $C\text{-}structs$ must have been proposed (*Non-triviality*); if a node decided a $C\text{-}struct$ 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 $C\text{-}struct$ (*Liveness*); and two $C\text{-}structs$ decided by two different nodes are prefixes of the same $C\text{-}struct$ (*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 on a node p_i, with the following semantics: the sequence of k consecutive calls of DECIDE(c_1) $\bullet$ DECIDE(c_2) $\bullet \cdots \bullet$ DECIDE(c_k) on p_i 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 $C\text{-}structs$ are still the same if they only differ by a permutation of non-conflicting commands.

IV. OVERVIEW OF CAESAR

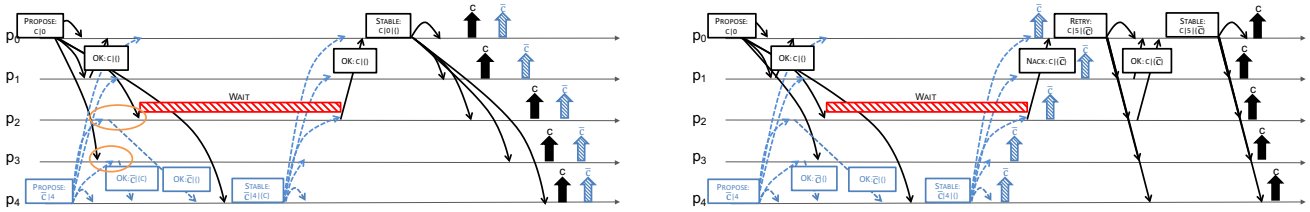
We introduce CAESAR incrementally by starting from a base protocol, which only provides reliable broadcast of commands, and then we present the design of the final protocol, which implements Generalized Consensus. We consider the first protocol as a reference point to show the minimal costs that are 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 V 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 [22].

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 V-A 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 V we explain that a fast quorum



(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 $\mathcal{T} = 0 < \bar{\mathcal{T}} = 4$.

is required at this stage due to the lower-bound defined in [22]. Here, we assume c's leader does not fail or is suspected; the case of faulty leaders is discussed in Section V-E.

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 $\mathcal{OK}$ 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 still 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 IV-A)? How does a command's leader behave if one of the nodes in the replying quorum rejects a proposed timestamp (Section IV-B)?

A. 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., $\bar{c}$.

B. 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 V).

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 IV-A. 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 V-C and V-F). 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.

V. 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 V-A, the procedure for a fast decision in Section V-B, the procedure for a slow decision in Section V-C, and the behavior of the protocol in case of failures in Section V-E. 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 V-D and detailed in the technical report [23].

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 for historical reasons. Finally, the pseudocodes of auxiliary functions and the recovery from a failure are provided in Figures 3 and 5, respectively.

A. 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 the *don't-care term* "$-$" 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.

B. 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 V-E. 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 IV-A. 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 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 4–8 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 1–3 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 $\mathcal{P}red_j$.

```
1:  function Set ComputePredecessors(c, Time, Whitelist)
2:      Pred_j ← {c̄ : c̄ ∼ c
                                  ∧
      (Whitelist = null ⇒ ∃⟨c̄, T̄, −, −, −, −⟩ ∈ H_j : T̄ < Time)
                                  ∧
      (Whitelist ≠ null ⇒ c̄ ∈ Whitelist ∨
          ∃⟨c̄, T̄, −, slow-pending/accepted/stable, −, −⟩ ∈ H_j :
                                                  T̄ < Time) }
3:      return Pred_j
4:  function Boolean Wait(c, Time)
5:      wait until ∀⟨c̄, T̄, Pred̄, −, −, −⟩ ∈ H_j,
                    (c̄ ∼ c ∧ Time ≤ T̄ ∧ c ∉ Pred̄ ⇒
                        ∃⟨c̄, T̄, Pred̄, accepted/stable, −, −⟩ ∈ H_j)
6:      if ∃⟨c̄, T̄, Pred̄, accepted/stable, −, −⟩ ∈ H_j :
                                  c̄ ∼ c ∧ Time < T̄ ∧ c ∉ Pred̄ then
7:          return NACK
8:      else return OK
9:  function BreakLoop(c)
10:     ⟨c, T, Pred, stable, B, ⊥⟩ ← H_j.Get(c)
11:     for all c̄ ∈ Pred : ⟨c̄, T̄, Pred̄, stable, B̄, ⊥⟩ ∈ H_j ∧ T̄ < T do
12:         H_j.Update(c̄, T̄, Pred̄ \ {c}, stable, B̄, ⊥)
13:     for all c̄ ∈ Pred : ⟨c̄, T̄, Pred̄, stable, B̄, ⊥⟩ ∈ H_j ∧ T̄ > T do
14:         Pred ← Pred \ {c̄}
15:     H_j.Update(c, T, Pred, stable, B, ⊥)
16: function Boolean Deliverable(c)
17:     return (c ∪ H_j.GetPredecessors(c)) ⊆ Decided_j
```

Fig. 3. Auxiliary functions - node p_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 [13], [10], 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 V-F), 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 V-F, 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

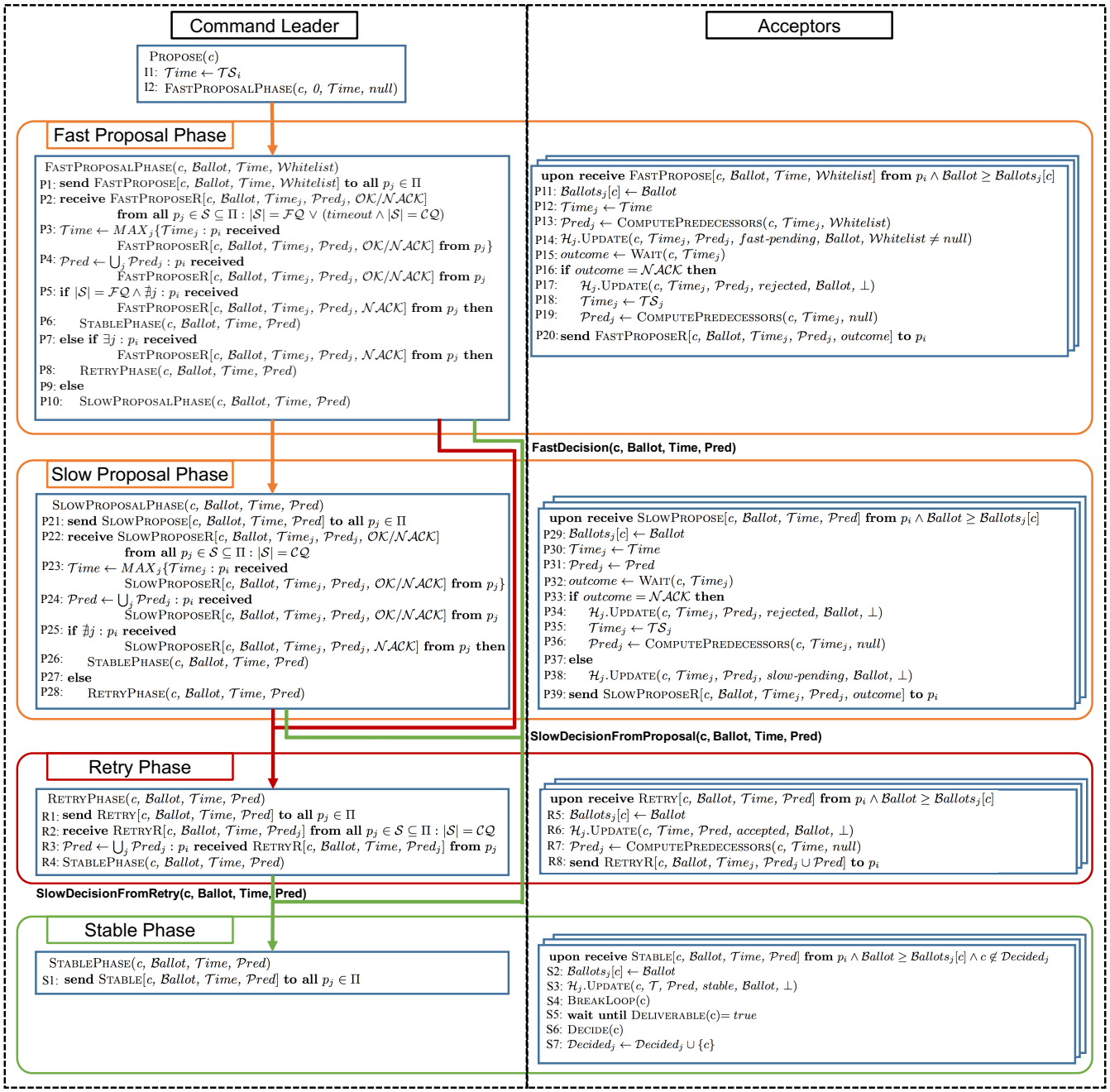


Fig. 4. CAESAR's pseudocode. The left part is executed by the command c's leader p_i, and the right part can be executed by any acceptor p_j (including p_i).

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.

C. 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 V-D.

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 guarantees that, even with f failures, there always exists a correct node that confirmed $\mathcal{T}ime$ 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 $\mathcal{P}red$ set in input to the next *stable phase* (line R3). This is because, as shown in Section IV-B, 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 V-F), 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, $\mathcal{T}ime$, and $\mathcal{P}red$, it only updates the tuple for c in its $\mathcal{H}_j$ by marking it as $accepted$ with $\mathcal{T}ime$ and $\mathcal{P}red$ (line R5), and it computes a new predecessors set $\mathcal{P}red_j$ 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 $\mathcal{P}red_j$ as well as the one previously received by the leader (line R8).

D. 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* (see lines P21–P39). 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 [23].

E. 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:    Ballots_k[c]++
3:    send RECOVERY[c, Ballots_k[c]] to all p_j ∈ Π
4:    receive RECOVERYR[c, Ballots_k[c],
                  ⟨c, T_j, Pred_j, −, B_j, ⊥/⊤⟩/NOP]
                  from all p_j ∈ S ⊆ Π : |S| = CQ
5:    MaxBallot ← MAX{B_j : p_i received
          RECOVERYR[c, Ballots_k[c], ⟨c, T_j, Pred_j, −, B_j, ⊥/⊤⟩] }
6:    RecoverySet ← {⟨p_j, T_j, Pred_j, −, ⊥/⊤⟩ : p_i received
          RECOVERYR[c, Ballots_k[c], ⟨c, T_j, Pred_j, −, B_j, ⊥/⊤⟩]
                  from p_j ∧ B_j = MaxBallot }
7:    if ∃ ⟨p_j, T_j, Pred_j, stable, ⊥⟩ ∈ RecoverySet then
8:      STABLEPHASE(c, Ballots_k[c], T_j, Pred_j)
9:    else if ∃ ⟨p_j, T_j, Pred_j, accepted, ⊥⟩ ∈ RecoverySet then
10:     RETRYPHASE(c, Ballots_k[c], T_j, Pred_j)
11:   else if ∃ ⟨p_j, T_j, Pred_j, rejected, ⊥⟩ ∈ RecoverySet then
12:     Time ← TS_i
13:     FASTPROPOSALPHASE(c, Ballots_k[c], Time, null)
14:   else if ∃ ⟨p_j, T_j, Pred_j, slow-pending, ⊥⟩ ∈ RecoverySet then
15:     SLOWPROPOSALPHASE(c, Ballots_k[c], T_j, Pred_j)
16:   else if |RecoverySet| > 0 then
17:     Time ← T_j :
            ∃⟨p_j, T_j, Pred_j, fast-pending, ⊥/⊤⟩ ∈ RecoverySet
18:     Pred ← ⋃_j Pred_j :
            ⟨p_j, T_j, Pred_j, fast-pending, ⊥/⊤⟩ ∈ RecoverySet
19:     if ∃ ⟨p_j, T_j, Pred_j, fast-pending, ⊤⟩ ∈ RecoverySet then
20:       WhiteList ← Pred
21:     else if |RecoverySet| ≥ ⌊CQ/2⌋ + 1 then
22:       WhiteList ← {c̄ ∈ Pred : ∄S ⊆ RecoverySet,
                                  |S| ≥ ⌊CQ/2⌋ + 1 ∧
              ∀⟨p_j, T_j, Pred_j, fast-pending, ⊥⟩ ∈ S, c̄ ∉ Pred_j }
23:     else
24:       WhiteList ← null
25:     FASTPROPOSALPHASE(c, Ballots_k[c], Time, WhiteList)
26:   else
27:     Time ← TS_i
28:     FASTPROPOSALPHASE(c, Ballots_k[c], Time, null)
29: upon receive RECOVERY[c, Ballot] from p_k ∧ Ballot > Ballots_j[c]
30:   Ballots_j[c] ← Ballot
31:   if H_j.CONTAINS(c) then
32:     send RECOVERYR[c, Ballots_j[c], H_j.GETINFO(c)] to p_k
33:   else
34:     send RECOVERYR[c, Ballots_j[c], 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. 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 V-B). *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).

F. Correctness

The complete formal proof on the correctness of CAESAR is in the technical report [23], where we have also formalized a description of the algorithm in TLA+ [24], which has been model-checked with TLC model-checker. Here we provide the main intuition on how we proceeded in proving that CAESAR implements the specification of Generalized Consensus.

Let us also define the predicate DECIDED[c,$\mathcal{T}$,$\mathcal{P}red$,$\mathcal{B}$] as a predicate that is equal to true whenever a node decides a command c with timestamp $\mathcal{T}$, predecessors set $\mathcal{P}red$, and ballot $\mathcal{B}$. Then we can prove that CAESAR guarantees *Consistency* by proving the following two theorems:

- $\forall c, \bar{c}$, (DECIDED[c,$\mathcal{T}$,$\mathcal{P}red$,$\mathcal{B}$] $\wedge$ DECIDED[$\bar{c}$,$\bar{\mathcal{T}}$,$\overline{\mathcal{P}red}$,$\overline{\mathcal{B}}$] $\wedge$ $\bar{\mathcal{T}} < \mathcal{T} \wedge c \sim \bar{c} \Rightarrow \bar{c} \in \mathcal{P}red$);
- $\forall c$ ($\exists \mathcal{B}$, 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[$c$,$\mathcal{T}'$,$\mathcal{P}red'$,$\mathcal{B}'$] $\Rightarrow \mathcal{T}' = \mathcal{T} \wedge \mathcal{P}red' = \mathcal{P}red$)).

VI. 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. As a result, by categorizing a workload with 10% of conflicting commands, we refer to the fact that 10% of the accessed keys belong to the shared pool. 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 (the caption indicates that).

We deployed the competitors on five nodes located in Virginia (US), Ohio (US), Frankfurt (EU), Ireland (EU), and

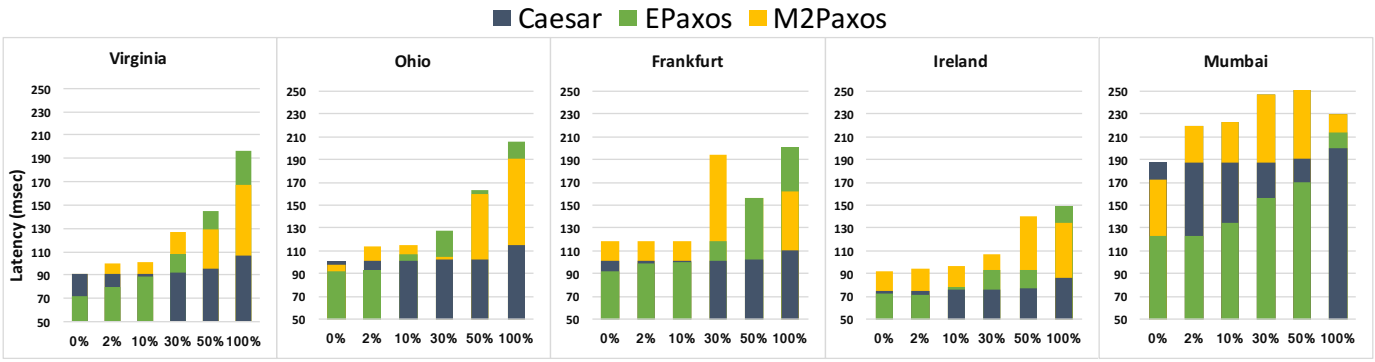


Fig. 6. Average latency for ordering and processing commands by changing the percentage of conflicting commands. Batching is disabled. Bars are overlapped: e.g., in the case of 30% conflicts in Virginia, latency values are 90 msec, 108 msec, and 127 msec, for CAESAR, EPaxos, and M^2Paxos, respectively.

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. In CAESAR, conflicting commands are tracked using a Red-Black tree data structure ordered by their timestamp.

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 distance to have a quorum of responses.

A. 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 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% – total order}.

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 conflicting commands 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 included also the case of 100% conflicts. Here all competitors behave poorly given the need for ordering all commands, which does not represent their ideal deployment.

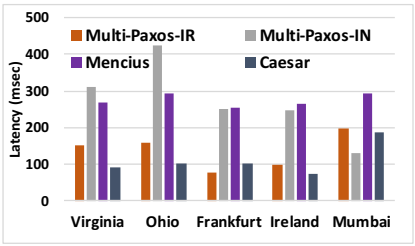


Fig. 7. Average latency for ordering commands of Multi-Paxos (with a close and faraway leader), Mencius, and CAESAR. Batching is disabled.

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 percentage of conflicting commands injected in the system. CAESAR 0% has also been included for reference. Mencius's performance is similar across the nodes because it needs to collect feedbacks from all consensus participants, and 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 with results in Figure 6, Multi-Paxos-IR and Multi-Paxos-IN are, on average, 5% and

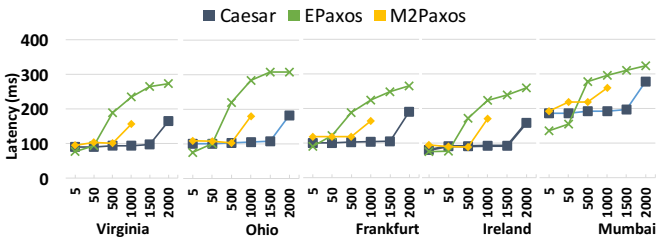


Fig. 8. Latency per node while varying the number of connected clients (5 – 2000). Network messages are not batched.

40% slower than CAESAR 100%, respectively.

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% conflicting commands. The x-axis indicates the total number of connected clients. The complex delivery phase of EPaxos, where it has to analyze 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 more than 1500 total clients are connected. M^2Paxos stops scaling after 1000 connected clients due to the impact of the forwarding mechanism.

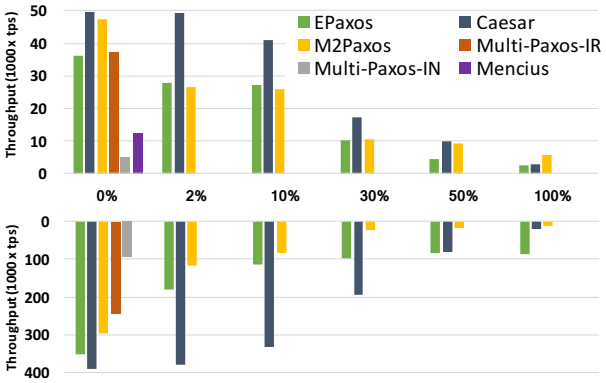


Fig. 9. Throughput by varying the percentage of conflicting commands. In the top part of the plot batching is disabled, in the lower part it is enabled.

Figure 9 shows the total throughput obtained by each competitor. Performance of Multi-Paxos and Mencius is placed under the 0% case. The upper part of the plot has network batching disabled. Here the performance of CAESAR degrades by only 17% when moving from no conflict to 10% of conflicting 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 10% conflicts. M^2Paxos is the system that behaves best when commands are 100% conflicting. 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.

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. At 50% and 100% of conflicting commands, 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 to 10% of conflicting commands by providing 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 it degrades its performance substantially if the leader moves to a faraway node.

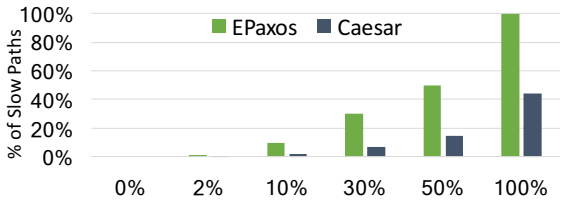
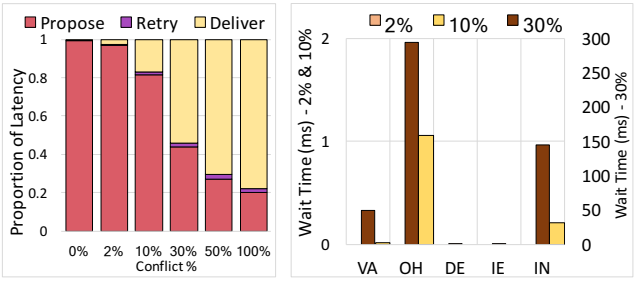


Fig. 10. % of commands delivered using a slow decision by varying % of conflicting commands. Batching here is disabled.

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%) 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.

In Figure 11, we report the internal statistics of CAESAR gathered during the experiment in Figure 9. Figure 11(a) shows the breakdown of the proportion of latency consumed by each ordering phase of the protocol. For no conflicts (0%, 2%), the maximum time is spent in the proposal phase. The cost



(a) Ordering phases. (b) Wait condition.

Fig. 11. Latency breakdown for CAESAR.

of the delivery is very low, since there are no dependencies. However, as conflicts increase, 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 11(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% of conflicting commands to highlight the difference with respect to the case of 2% and 10%. 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.

B. Recovery

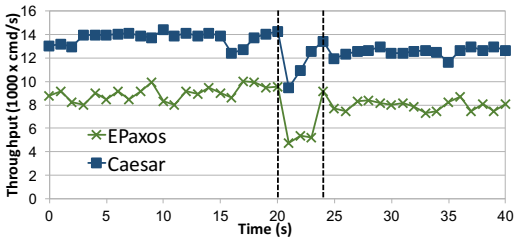


Fig. 12. Throughput when one node fails.

In Figure 12, 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.

VII. CONCLUSION

This paper shows that existing high-performance implementations of Generalized Consensus suffer from performance degradation when the percentage of conflicting commands increases. The reason is related to the way they establish a fast decision. In this paper we present an innovative technique that provides a very high probability of fast delivery.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable comments. This work is partially supported by Air Force Office of Scientific Research under grant FA9550-15-1-0098 and by US National Science Foundation under grant CNS-1523558.

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, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally Distributed Database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.
[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, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "MDCC: Multi-data Center Consistency," in *EuroSys*, 2013, pp. 113–126.
[7] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi, "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] "Google Cloud Spanner - https://cloud.google.com/spanner/."
[10] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is More Consensus in Egalitarian Parliaments," ser. SOSP, 2013, pp. 358–372.
[11] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building Efficient Replicated State Machines for WANs," ser. OSDI, 2008, pp. 369–384.
[12] A. Turcu, S. Peluso, R. Palmieri, and B. Ravindran, "Be General and Don't Give Up Consistency in Geo-Replicated Transactional Systems," ser. OPODIS, 2014, pp. 33–48.
[13] L. Lamport, "Generalized Consensus and Paxos," Microsoft Research, Tech. Rep. MSR-TR-2005-33, March 2005.
[14] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran, "Making fast consensus generally faster," in *DSN*, 2016, pp. 156–167.
[15] L. Lamport, "Fast Paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.
[16] P. Sutra and M. Shapiro, "Fast Genuine Generalized Consensus," ser. SRDS, 2011, pp. 255–264.
[17] W. Wei, H. T. Gao, F. Xu, and Q. Li, "Fast mencius: Mencius with low commit latency," in *IEEE INFOCOM*, 2013, pp. 881–889.
[18] J. Du, D. Sciascia, S. Elnikety, W. Zwaenepoel, and F. Pedone, "Clock-RSM: Low-Latency Inter-datacenter State Machine Replication Using Loosely Synchronized Physical Clocks," ser. DSN, 2014, pp. 343–354.
[19] 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.
[20] 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.
[21] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Springer, 2006.
[22] L. Lamport, "Future directions in distributed computing," A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, Eds., 2003, ch. Lower Bounds for Asynchronous Consensus, pp. 22–23.
[23] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran, "Speeding up Consensus by Chasing Fast Decisions," Tech. Rep., 2017. [Online]. Available: https://arxiv.org/abs/1704.03319
[24] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.