A Low-latency Consensus Algorithm for Geographically Distributed Systems

Balaji Arun

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Haibo Zeng
Robert Broadwater

February 28, 2017
Blacksburg, Virginia

Keywords: Distributed Systems, Fault Tolerance, State Machine Replication, Multi-Leader Consensus
Copyright © 2017, Balaji Arun

A Low-latency Consensus Algorithm for Geographically Distributed Systems

Balaji Arun

(ABSTRACT)

This thesis presents Caesar, a novel multi-leader Generalized Consensus protocol for geographically replicated systems. Caesar is able to achieve near-perfect availability, provide high performance - low latency and high throughput compared to the existing state-of-the-art, and tolerate replica failures. Recently, a number of state-of-the-art consensus protocols that implement the Generalized Consensus definition have been proposed. However, the major limitation of these existing approaches is the significant performance degradation when application workload produces conflicting requests. Caesar's main goal is to overcome this limitation 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. It only switches to a slow decision if there is no chance to agree on the proposed order for that request. Caesar is able to achieve this using a combination of wait condition and logical timestamping. 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 presence of 30% conflicting requests, and single-leader (e.g., Multi-Paxos) by as much as 3.5x. The protocol is also resistant to heavy client loads unlike existing protocols.

This work is supported in part by Air Force Office of Scientific Research (AFOSR) under grant FA9550-15-1-0098 and by National Science Foundation (NSF) under grant CNS-1523558.

A Low-latency Consensus Algorithm for Geographically Distributed Systems

Balaji Arun

(GENERAL AUDIENCE ABSTRACT)

Today, there exists a plethora of online services (e.g. Facebook, Google) that serve millions of users daily. Usually, each of these services have multiple subcomponents that work cohesively to deliver a rich user experience. One vital component that is prevalent in these services is the one that maintains the shared state. One example of a shared state component is a database, which enables operations on structured data. Such shared states are replicated across multiple server nodes, and even across multiple data centers to guarantee availability, i.e., if a node fails, other nodes can still serve requests on the shared state; low-latency, i.e., placing the copy of the shared state in a datacenter closer to the users will reduce the time required to serve the users; and scalability, i.e., the bottleneck that a single server node cannot serve millions of concurrent requests can be alleviated by having multiple nodes serve users at the same time. These replicated shared states need to be kept consistent i.e. every copy of the shared state must be the same in all the replicated nodes, and maintaining this consistency requires that each of these replicating nodes communicate with each other and reach an agreement on the order in which the operations on the shared data should be applied. In that regard, this thesis proposes Caesar, a consensus protocol with the aforementioned guarantees that will ease the deployment of services that contain a shared state. It addresses the problem of performance degradation in existing approaches when the same part of the shared state are accessed by multiple users that are connected to different server nodes. The effectiveness of Caesar is demonstrated through an evaluation study performed by deploying the protocol on five of Amazon's data centers around the world. Caesar outperforms the existing state-of-the-art by as much as 3.5x. Caesar is also resistant to heavy client loads unlike existing protocols.

Dedication

To my parents and Lekha.

Acknowledgments

I would like to acknowledge the following people who made this thesis possible:

My foremost thank you to Dr. Binoy Ravindran for recognizing my potential and aspirations for Graduate study, and advising me throughout towards achieving this feat. I also thank Dr. Ravindran for introducing me to Dr. Roberto Palmieri and Dr. Sebastiano Peluso, with whom I closely worked since my undergraduate days.

I cannot thank Dr. Roberto Palmieri enough for teaching me the fundamentals of distributed systems and also lending a hand whenever I needed one - be it understanding an algorithm or implementing good code. I could not have accomplished this thesis without him.

I am fortunate to have worked closely with Dr. Sebastiano Peluso in this thesis. His tremendous knowledge made this work possible. I heartily thank him for sharing his knowledge throughout the work, brainstorming ideas when problems arose, and for taking the time to read my thesis and suggesting changes.

Thanks to the current and past members of System Software Research Group for their support, skill and friendship: Giuliano, Sachin, Utkarsh, Masoomeh, Chris, Anthony, Rob, Jack, Cindy, Josh, Ian, Sandeep, Marina, Ahmed, Mohamedin, Alex, Saif, Wen, and Sean.

My dad has always ensured that I make the right decisions throughout my life and his perseverance that I pursue Graduate study has been nothing less than rewarding. My mom encouraged me to always pursue my dreams. My grandparents were part of most of my childhood days, and their love and affection has always been a motivation to pursue my dreams. I dedicate this thesis to all of them.

My girlfriend, Lekha, has been patiently waiting all these years to let me pursue my education. The day I defend this thesis marks 5 years since we last met, and that just tells everything.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Contribution 2

1.3 Summary of Contribution 2

1.4 Thesis Organization 3

2 Background and Related Work **4**

2.1 Replication 4

2.2 Consensus 5

2.3 Generalized Consensus 8

3 System Model and Overview **12**

3.1 System Model 12

3.2 Overview of Caesar 13

3.2.1 Base Protocol 13

3.2.2 Caesar 14

4 Protocol Details **18**

4.1 Data Structures per node p_i 19

4.2 Fast Decision 20

4.3 Slow Decision 24

4.4 Unavailability of Fast Quorums 25

4.5 Recovery from Failures . . . 26

5 Implementation and Evaluation **29**

5.1 Implementation . . . 29

5.2 Evaluation . . . 30

5.2.1 Non-faulty Scenarios . . . 31

5.2.2 Recovery . . . 35

6 Correctness **37**

6.1 Overview . . . 37

6.2 Terminology . . . 38

6.3 Proof on Consistency . . . 38

7 Conclusion and Future Work **47**

7.1 Conclusion . . . 47

7.2 Lessons Learnt . . . 48

7.3 Future Work . . . 48

Bibliography **49**

List of Figures

2.1 A Paxos Consensus Instance 7

3.1 Fault-tolerant broadcast execution vs. CAESAR execution 14

3.2 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 $\mathcal{T} = 0 < \bar{\mathcal{T}} = 4$. 16

4.1 Auxiliary functions - node p_j 22

4.2 Pseudocode of CAESAR. 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). . . . 23

4.3 RECOVERY phase executed by node p_k. Node p_j is a receiver of the RECOVERY message. 28

5.1 Average latency for ordering and processing commands by changing the percentage of conflicting commands. Batching disabled. Given a percentage value, bars of competitors are overlapped: e.g., in the case of 30% conflicting commands and the node in Virginia, latency values are 90 msec, 108 msec, and 127 msec, for CAESAR, EPaxos, and M^2Paxos, respectively. 30

5.2 Average latency for ordering commands of Multi-Paxos (with a close and faraway leader), Mencius, and CAESAR. Batching is disabled. 32

5.3 Latency per node while varying the number of connected clients (5 – 2000). Network messages are not batched. 32

5.4 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. 34

5.5 % of commands delivered using a slow decision by varying % of conflicting commands. Batching here is disabled. 35

5.6 Latency breakdown for Caesar. 35
5.7 Throughput when one node fails. 35

List of Tables

5.1 Average round trip latencies (in ms) between sites 31

Chapter 1

Introduction

Today's online services are expected to reach global customers. For instance, Gmail and Facebook each have over one billion monthly active users [1, 2]. Although these large enterprises largely fueled their growth with the help of their own infrastructure, there are a multitude of enterprises that leverage the cloud to quickly build large scale services and reach millions, and even billions, of users. For example, Amazon.com and Netflix leverage Amazon Web Services [3] to reach their customers globally [4, 5]. In order to facilitate such large scale services, developers rely on *replication*. Replication is perhaps the most viable solution to increase availability and scalability of applications. Developers use replication to duplicate both stateful (e.g. databases) as well as stateless (e.g. application servers) services. The most challenging is stateful replication, which requires coordination among replicas to maintain a strongly consistent copy of the shared state across all replicas. Maintaining consistent copies makes stateful replication transparent to the end user, and thus applications can simply be programmed to use the replicated system as if it were a non-replicated one. The cost of the coordination is exaggerated in geographically replicated (geo-scale) services where replicas are spread across geographic locations and connected via high latency links. In order to decrease the coordination cost, weaker consistency policies have been proposed [6, 7], but they increase application development complexity. The contribution of this thesis addresses the challenge of geographical replication by providing a novel Consensus (agreement) protocol that provides both strong consistency and low latency in the wide area.

1.1 Motivation

There exists a bunch of techniques to replicate stateful services. Amongst, State Machine Replication is a popular replication technique that has been used extensively within a data center, to mask machine failures, and among geo-replicated sites, to mask datacenter failures. Regardless of the deployment scenario, synchronization mechanisms are required to

maintain strongly consistent copy of the shared state across all replicas, thus easing the development of replicated services. These mechanisms, known as Consensus (agreement)[8] protocols, decide the order in which operations will be executed by every replica in the system. However, existing Consensus implementations are not apt for geo-replication, because they do not provide two – low-latency and high-throughput – of the four desirable properties for building today's services – low client-perceived latency, high-throughput, availability and strong consistency.

Most existing approaches [9, 10] perform two round trip time (RTT) of communication to agree on the order for an operation. In geo-replication, two communication RTT significantly increases the client-perceived latency as replicas communicate via high-latency links (such as internet). Some other approaches [11, 12, 13] are able to reach consensus on operations in one RTT, but under a particular scenario: the case when two replicas operate on different parts of the shared state (e.g. different tables in database) at any given time. This scenario constitutes a non-conflicting workload. On the other hand, under conflicting workloads, that is when replicas operate on the same part of the share state concurrently, these approaches incur high-latency due to multiple RTTs involved to reach agreeement and provide low throughput due to the complex computations that they perform.

1.2 Thesis Contribution

This thesis focuses on improving the current state, specifically the performance aspects, of Consensus algorithms for geo-scale deployments. The contribution of this thesis are:

- The design and implementation of CAESAR, a Consensus protocol designed for geo-scale deployments that is able to maintain 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).
- The performance evaluation of CAESAR and a comparison to that of existing state of the art.

1.3 Summary of Contribution

This thesis presents the theoretical and implementation details of CAESAR. CAESAR solves consensus by taking into account the various scenarios that occur when messages are delivered asynchronously. At first, this thesis highlights these scenarios and explain how CAESAR's novelties enable it to order a command in one RTT despite the presence of concurrent conflicting commands. Details on how CAESAR inherits the advantages of existing approaches

– Mencius [14] and EPaxos [11] – and overcomes the inherent pitfalls in these existing approaches is presented. In summary, CAESAR can agree on the order for an operation in one RTT, called *fast decision*, when the workload is non-conflicting. Under conflicting workloads, CAESAR maximizes the number of fast decisions using a novel *wait condition*. Under scenarios where a fast decision is not possible despite the presence of the wait condition, one more RTT is taken to decide on the order for the operation, and this is called *slow decision*.

CAESAR has been implemented in Java and evaluated using key-value store interfaces. Using the key-value store interfaces, different workloads were injected by varying the percentage of conflicting commands and measure various performance parameters. CAESAR is contrasted against: EPaxos [11] and M^2Paxos [13], multi-leader quorum-based Generalized Consensus implementations; Mencius [14], a multi-leader timestamp-based Consensus implementation that does not rely on quorums; Multi-Paxos [9], a single-leader Consensus implementation. For evaluation, the systems were deployed using the Amazon EC2 infrastructure in 5 geo-replicated sites.

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×. These performance boosts are due to the higher percentage of fast decisions accomplished. For example, at 30% of conflicting workload, CAESAR takes up to 70% fewer slow decisions compared to EPaxos.

1.4 Thesis Organization

This thesis is organized as follows.

- Chapter 2 provides the background for CAESAR, the contribution of this thesis, and discusses the related previous works in the space.
- Chapter 3 motivates the design ideas behind CAESAR.
- Chapter 4 delves into the details of the protocol including the data structures and algorithm pseudocode.
- Chapter 5 provides an comprehensive evaluation of CAESAR
- Chapter 6 provides the formal proof of correctness for CAESAR.
- Chapter 7 concludes this thesis and offers future work to extend the contribution of this thesis.

Chapter 2

Background and Related Work

This chapter motivates the need for replication and overviews different replication techniques in Section 2.1, before diving into the details of the problem of distributed consensus in Section 2.2. Section 2.3 presents Generalized Consensus, a variant of consensus that parallelizes the agreement process by exploiting the application semantics. In Sections 2.2 and 2.3, some of the important consensus algorithms that relate to Caesar, the contribution of this thesis, are discussed and contrasted.

2.1 Replication

Replication is a fundamental technique that is widely adopted to implement fault-tolerance and improve the performance and scalability of today's online services. Performance is usually measured in terms of the throughput and the client perceived latency. As more and more services strive to reach customers globally, reducing the client perceived latency is of prime concern. To accomplish this, replicas are placed close to the clients. However, geographical scale replication is a challenge due to the high latency perceived between replicas. Many proposals in literature, such as [14, 11] and the contribution of this thesis, address this challenge. All the proposed replication techniques can be grouped into two broad categories [15]: Primary Copy and Update Everywhere.

In Primary Copy (aka. Passive) Replication, a master replica primarily serves client requests, while a set of slave replicas that synchronously/asynchronously receive the master replica's state and update their state. Most enterprise-grade databases today provide this feature out of the box [16, 17]. The disadvantages of this technique are twofold: (a) the master replica crash leads to the system's temporary unavailability until one of the slave replicas is elected as the master; (b) when the replication is asynchronous, the slave replicas do not necessarily reflect the master's most recent state, and thus when the master fails, the newly elected master will serve requests from an older state and requires manual intervention to correct

the system state.

On the other hand, in Update Everywhere replication, there is no distinction between a primary and a copy replica. Thus, every node in the distributed system is responsible for executing client requests. One of the most widely adopted and studied Update Everywhere replication technique is State Machine Replication (aka. Active Replication) [18].

Under the state-machine approach, a distributed system is modeled as a set of states, and transition among states happen deterministically by executing client requests. Every replica in a distributed system must implement a state machine and have to be initialized with the same initial state. Thus, when the same sequence of client requests are executed on every replica, each one of them will reach the same final state. This technique is very effective in masking failures in a distributed system as every command issued to the system is executed by each replica. Therefore, if a replica fails (by crashing), the system does not fail because each replica maintains its own copy of the shared state and thus can make progress.

Maintaining a consistent copy of the state across all replicas entails the implementation of Consensus. The rest of the chapter focuses on Consensus and related algorithms, since the contribution of this thesis addresses the problem of Consensus.

2.2 Consensus

Every replica of a distributed system is subject to numerous concurrent client requests. In order to keep the system consistent, replicas have to coordinate and agree on client requests that each node should execute, even in the presence of faults. This is widely known as the consensus problem. A typical consensus problem is defined by the following safety requirements [19]:

- *Nontriviality*: A request can only be agreed upon if it was proposed by a client.
- *Stability*: Once a request has been agreed for execution, no replica can revert its decision.
- *Consistency*: Two different replicas cannot agree on different request for execution.

Many consensus algorithms have been proposed in literature; the most popular being Paxos [10]. Works such as Multi-Paxos, Mencius, and Generalized Paxos and its derivatives including the contribution of this thesis have been inspired by Paxos. The next subsections provide an overview of Paxos and some of its close derivatives. Generalized Consensus and related protocols are discussed in Section 2.3.

Paxos

Paxos [10] is the earliest, widely accepted and studied consensus algorithm. Many distributed systems in production such as Spanner [20] use Paxos or its variant under the hood. Each replica in a Paxos-based distributed system runs its own instance of the Paxos protocol. Replicas communicate with each other using messages. Each instance of the protocol performs any/all of the following roles: proposer, acceptor, learner. The proposer proposes values (a client request) that a learner should learn (execute). The acceptor decides the value to be learnt. Multiple acceptors are used for fault-tolerance.

When a value needs to be chosen, the proposer acquires ownership of the value, assigns it a proposal number, and proposes it to a group of acceptors. An acceptor may or may not accept the proposal as explained below. The proposer is notified about the acceptor's decision. If the proposer receives a positive acknowledgement from a majority of the acceptors, then the proposer sends a message to the learners to learn the value. This entire process of choosing a value constitutes a Consensus Instance. A Consensus Instance should be executed for each value to be decided.

Each Consensus Instance consists of three phases, as shown in Figure 2.1.

Phase 1: This phase involves a proposer seeking a promise from a majority of acceptors for accepting a value that the proposer will propose in the next phase.

(a) A proposer chooses a proposal number n and sends a prepare message with n to a majority of acceptors.

(b) An acceptor upon receiving the `prepare` request for proposal number n, which is greater than that of any proposal number the acceptor received so far (for the consensus instance), responds with a promise not to accept any proposals with proposal number less than n along with the highest-numbered proposal (i.e. value v, if any) that it has accepted.

Phase 2: This phase involves a proposer trying to seek acceptance from a majority of the acceptors to accept the proposal number n, prepared during the previous phase, with a value v.

(a) Upon receiving responses to its `prepare` request from a majority of acceptors, the proposer then sends an `accept` message to all the acceptors with proposal number n and value v, where v is the value of the highest-numbered proposal among the responses if any, or is the value that the proposer wants to choose in that instance.

(b) Whenever an acceptor receives an `accept` request for proposal n with value v, it will `accept` the proposal if it did not promise any other `prepare` request with proposal number higher than n. Otherwise, it will `reject` the proposal.

Phase 3: Upon receiving acceptances from a majority of acceptors, the proposer will then send a learn message to the learners in order to learn the value v with proposal number n. If there is a reject, the proposer will retry from Phase 1 with a higher proposer number than n,

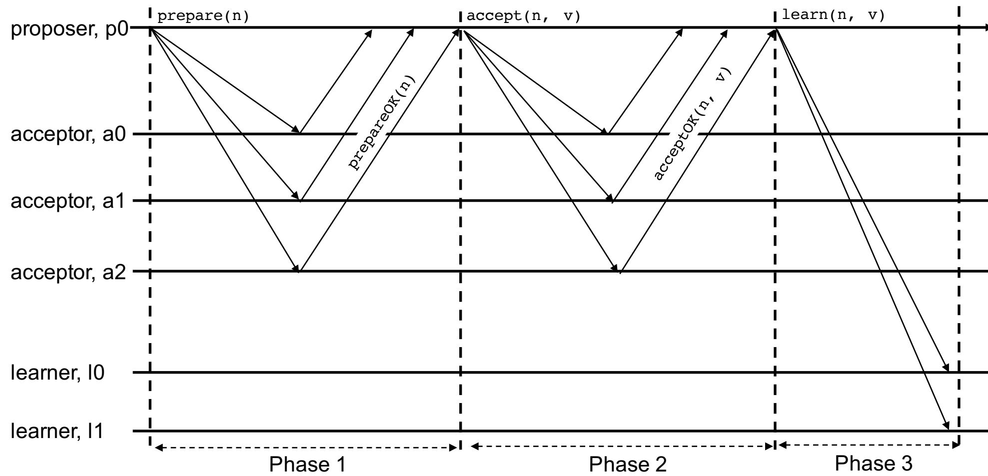


Figure 2.1: A Paxos Consensus Instance

if no value has been accepted in the consensus instance. Otherwise, the proposer will start a new Consensus Instance.

Optimization: Phase 3 can be removed by letting the acceptor in Phase 2b send the acceptance message directly to all the learners in addition to the proposer.

Multi-Paxos

The Paxos [10] algorithm makes certain that only a single value is chosen from those that have been proposed. However, the algorithm, as it is, does not guarantee progress. Consider the following scenario. A proposer p completes Phase 1 of the algorithm with proposal number n_1. Later, another proposer q chooses a higher proposal number $n_2 > n_1$ completes Phase 1. Thus, when proposer p seeks acceptance for a value with proposal number n_1, the acceptors will reject. Then, proposer p will restart from Phase 1 by choose a proposal number $n_3 > n_2$. Now, when proposer q runs Phase 2 of the algorithm for n_2, it will be rejected and it will rerun Phase 1 again with a higher proposal number than n_3. This can keep recurring stagnating the protocol's progress. Moreover, this phenomenon may occur during any consensus instances.

To overcome this problem, a distinguished proposer that is responsible for proposing values in all consensus instances is elected. When the distinguished proposer fails, another one is elected. In order to elect a distinguished proposer, the proposers run Phase 1 of the algorithm. Whenever a proposer sends a `prepare` with a proposal number n which is less than the highest proposal number ever agreed by an acceptor, the acceptor sends a explicit reject back to the proposer. The proposer will then cease sending `prepare` requests until the next proposer election cycle. This version of Paxos that guarantees progress is Multi-Paxos.

Mencius

Mencius [14] is a multi-leader consensus protocol that alleviates the drawbacks of Multi-Paxos. In Multi-Paxos, the performance of the entire system depends on the distinguished proposer (i.e. leader), since it is the only one responsible for proposing requests. With N replicas, the leader must handle $\theta(N)$ messages for each proposal sent, while the other replicas handle only $O(1)$. In addition, when the leader fails, the failure detector must kick in and notify the rest of the nodes so as to elect the next leader. During this time period, no values can be chosen, causing temporary unavailability of the system. Instead, in Mencius, the leadership is distributed among nodes in the system. More precisely, the consensus instances are equally distributed among nodes, and each node acts as the distinguished proposal for its consensus instances. For instance, a Replica R_{id} is the leader for all consensus instances i such that $(i \bmod N) = R_{id}$. This way, the load is equally spread across all the nodes in the system.

Shortcomings: Since the consensus instance space is pre-partitioned among replicas, every replica should decide commands in order for the system to make progress. For instance, when a replica R_1 decides a request B in instance 1, it should wait for replica R_0 to inform its decision for instance 0. Only when R_0 decides a request A in instance 0, R_1 can they go ahead and execute A followed by B. Thus, with Mencius, the system runs as fast as the slowest replica. Moreover, if a replica fails, no other replica can make progress until the failure is suspected and `no-ops` are proposed for the failed replica's consensus instances. To alleviate the problem of slow nodes, Fast Mencius has been proposed [21]. 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 (i.e. executing).

The contribution of this thesis, CAESAR, is also multi-leader protocol, like Mencius, However, CAESAR uses a combination of dependency gathering, as in [11], and a novel technique to allocate consensus instances to nodes in order to facilitate a quorum-based decision and delivery of commands. Thus, it does not suffer the slow node bottleneck.

2.3 Generalized Consensus

Protocols such as Paxos, Multi-Paxos and Mencius accept concurrent requests, agree on a common order to execute the requests, and execute the requests sequentially conforming to the order. The sequential execution severely limits the scalability of the system when requests that do not interfere are proposed concurrently. In contrast, Generalized Consensus allows concurrent requests to be ordered in any way as long as they do not interfere, thus multiple non-interfering requests can be executed concurrently.

According to the definition of Generalized Consensus [19], each node can propose a com-

mand c via PROPOSE(c) interface, and nodes decide command structures *C-struct* cs via DECIDE(cs) interface. Generalized Consensus is defined using the following properties:

- *Non-triviality*: Commands that are included in decided *C-structs* must have been proposed.
- *Stability*: 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.
- *Liveness*: If c has been proposed then c will be eventually decided in some *C-struct*.
- *Consistency*: Two *C-structs* decided by two different nodes are prefixes of the same *C-struct*.

There have been numerous proposals that implement Generalized Consensus, including EPaxos, Alvin, and M^2Paxos . Each one is overviewed in the following subsections.

EPaxos

EPaxos [11] is a multi-leader consensus protocol that, unlike Mencius, exhibits high availability despite replica crashes as long as a majority of the nodes are up and running. When a command arrives at an EPaxos replica, the replica takes ownership of the command (i.e., becomes the command leader) and goes through the following phases.

Phase 1: The command leader proposes its command c, after assigning it a sequence number, in a `PreAccept` message to all the other nodes in an attempt to commit (i.e. learn) c in the fast path. Any other nodes upon receiving the `PreAccept` message will return a set of dependencies for c back to the command leader in a `PreAcceptOK` message. Every node holds information about the commands that it has seen and uses this information to generate a list of commands that interfere/conflict with c (i.e. the dependency set).

The command leader upon receipt of `PreAcceptOK` messages from a fast quorum of replicas, checks whether the dependency sets returned are all the same. If so, it can execute Phase 3 right away and commit c; this is known as the fast phase. Otherwise, it executes Phase 2 - the slow phase.

Phase 2: The command leader for c combines the dependency sets it received in Phase 1 and sends a `Paxos-Accept` message to the other nodes. When a node receives a `Paxos-Accept` message, it then marks command c as *accepted* in its local history for the command, and replies to the command leader with a `AcceptOK` message.

The command leader yields until it receives a quorum of `AcceptOK` messages for c, and then it go ahead and runs Phase 3.

Phase 3: In this phase, the command leader sends a commit message to all the replicas and to the client. The replicas upon receiving the commit message will change the command c's status to committed. At this stage, a command c is committed with a dependency set *deps* and a sequence number *seq*.

In EPaxos, the ordering is detached from the command execution. There is a separate graph-based dependency linearization mechanism that is adapted to define the final order of execution of commands. As soon as a command c is committed, every replica builds a dependency graph by adding c and its dependencies. The next step is finding the strongly connected components and sorting them in reverse topological order. Once the commands in each strongly connected components are sorted according to their sequence number *seq*, they are executed one by one.

EPaxos employs dependency tracking and fast quorums to deliver non-conflicting commands using a fast path. In the presence of conflicts however, the protocol takes slow paths and uses four communication delays before deciding a command. 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.

Caesar also uses a dependency collection mechanism similar to EPaxos, but avoids the expensive graph processing phase by incorporating the linearization phase within the protocol's critical path, all without incurring any additional overhead. Moreover, even under the presence of conflicts, Caesar's novel fast decision scheme is optimized to increase the probability to decide in two communication delays even in those scenarios.

Alvin

Alvin [12] is also a multi-leader consensus protocol, but, unlike EPaxos, it is able to avoid the expensive computation on the dependency graphs enforced by EPaxos via a decision slot-centric approach *à la* Mencius. Alvin decides the order of a request after two communication delays under no conflicts. However, it still suffers from the same vulnerability to conflicts of EPaxos: a command's leader is not able to decide the command on a fast path if it observes discordant opinions about the command from a quorum of nodes, and thus incurs two more communication delays (four, in total) to decide. Caesar overcomes this problem as mentioned previously.

M^2Paxos

M^2Paxos [13] is a recently proposed implementation of multi-leader generalized consensus. Unlike other multi-leader protocols where nodes take ownership of the command proposed, M^2Paxos replicas take ownership of the objects accessed by the command. When a command c arrives at an M^2Paxos replica, the replica computes the owner(s) for the object(s) accessed

by the command and one of the following actions are taken depending on the result:

- *No replica owns the object(s)*: The node with the command runs Phase 1 of the Paxos protocol in order to acquire ownership of the object. If there are multiple objects, multiple Phase 1's are necessary.
- *Some other replica owns the objects(s)*: (a) If the object(s) is/are owned by another replica, then the command is simply forwarded to the replica and await its reply. (b) If multiple replicas own different objects, then the replica tries to acquire the ownership of the object by running Phase 1 of Paxos protocol
- *A replica owns the object(s)*: Whenever a command arrives at a node (either directly or by forwarding) that owns the object(s) for that command, it run Phase 2 and 3 of the Paxos protocol to commit the command. The command is then executed and the client notified.

M^2Paxos is able to combine the desirable features of supporting fast decisions, adopting only quorums with minimal size (i.e., a majority), and avoiding the exchange of dependencies among commands. The ownership acquisition phase for commands guarantees that a node having the ownership can autonomously take decisions on its commands. However, the scheme is optimized to deliver the lowest latency only in its favorable cases of negligible inter-node conflict rates. In case there are multiple nodes that compete for the decision of conflicting commands, the decision process must involve a forwarding phase or a ownership reacquisition phase, both of which contribute to high latency.

Chapter 3

System Model and Overview

This chapter begins the discussion of CAESAR with a description of the system model. Then, the intuition behind the protocol follows. To ease understanding, the protocol is described incrementally starting from a protocol that only provides reliable delivery of messages. CAESAR is then built on top of this base protocol.

3.1 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. Moreover, messages may experience arbitrarily long (but finite) delays.

Because of FLP [22], we assume that the system can be enhanced with the weakest type of unreliable failure detector [23] that is necessary to implement a leader election service [24]. In addition, due to the result in [8], we assume that at least a strict majority of nodes, i.e., $\lfloor \frac{N}{2} \rfloor + 1$, is correct and thus at most $f = \lceil \frac{N}{2} \rceil - 1$ nodes can be faulty at any time. We name *classic quorum*, or more simply *quorum*, any subset of Π with size at least equal to $\mathcal{CQ}$. We name *fast quorum* any subset of Π with size at least equal to $\mathcal{FQ}$. A fast quorum of nodes is required to have fast decisions in two communication delays, while classic quorum of nodes is required in case the protocol needs more than two communication delays to reach a decision. A classic quorum is also enough when there exists a single (per object/system) leader, like [10, 13]. Due to the lower-bound on asynchronous consensus that supports fast decision [25], we must require that any two fast quorums and a classic quorum always intersect, and therefore we choose $\mathcal{CQ} = \lfloor \frac{N}{2} \rfloor + 1$, and $\mathcal{FQ} = \lceil \frac{3N}{4} \rceil$ by minimizing $\mathcal{CQ}$.

We follow the definition of Generalized Consensus from [19], where each node can propose a command c via PROPOSE(c) interface, and nodes decide command structures *C-struct cs*

via DECIDE(cs) interface. The specification ensures *Non-triviality*, *Stability*, *Liveness*, and *Consistency*. 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 should be noted that, as specified in [19], two *C-structs* are still the same if they only differ by a permutation of non-conflicting commands.

3.2 Overview of Caesar

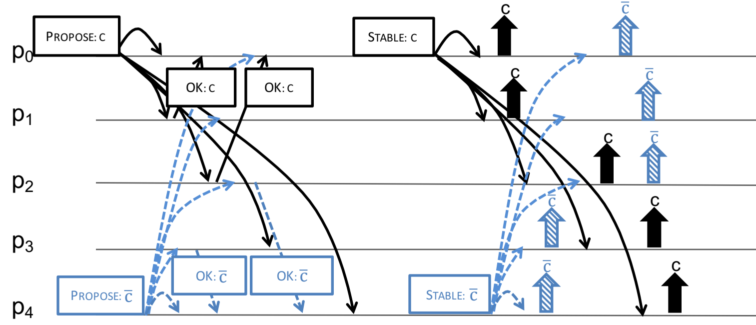
This section introduces CAESAR starting from a base protocol that provides a fault-tolerant broadcast of commands. This base protocol motivates the ideas behind CAESAR providing examples of different execution scenarios. The design of the final protocol, which implements Generalized Consensus, then follows. The first protocol is considered as a reference point to show the minimal costs that are required to implement our specification of Consensus and explain how CAESAR is able to maximize the probability to execute like the reference protocol. Chapter 4 provides all the details of CAESAR.

A necessary condition for implementing both a fault-tolerant 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 other correct node. This is because both the base protocol and CAESAR have to ensure that, whenever a command is executed by a node and the result externalized to clients, the command is durable in the system despite the crash of f nodes.

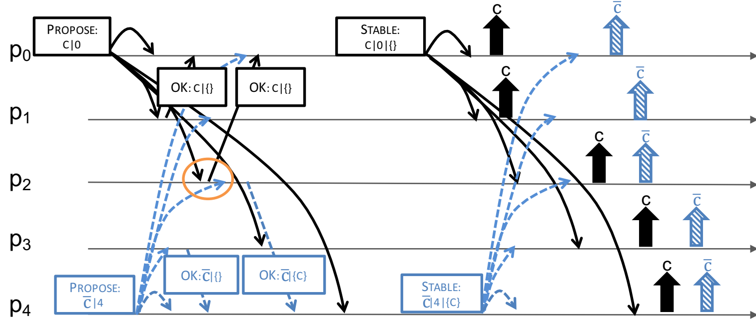
3.2.1 Base Protocol

The base protocol executes as shown in Figure 3.1(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 3.1(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; if c's leader crashes, there always exists at least one correct node that received the PROPOSE



(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.

Figure 3.1: Fault-tolerant broadcast execution vs. Caesar execution

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 number of network interactions required to implement consensus in an asynchronous system [25].

3.2.2 Caesar

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 3.1 for showing the intuition. Command c is associated with a unique logical *timestamp* $\mathcal{T}$ (see Section 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 Chapter 4 we explain that a fast quorum is required at this stage due to the lower-bound defined in [25]. Here, we assume c's leader does not fail or is suspected; the case of faulty leaders is discussed in Section 4.5.

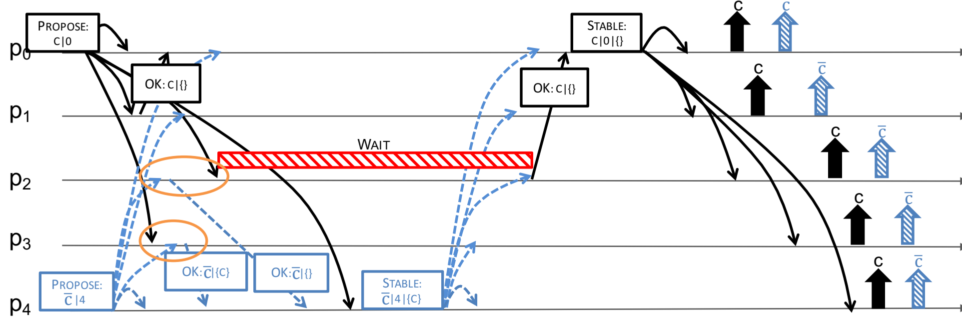
Figure 3.1(b) shows how CAESAR applies this idea to the execution of Figure 3.1(a). Node p_0 broadcasts c by proposing it with timestamp 0; then a quorum of nodes confirms c since none of those nodes has already received $\bar{c}$ with a timestamp greater than 0. The confirmation from a process p_j is 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 3.1(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 showed Figure 3.1(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 3.1(b).

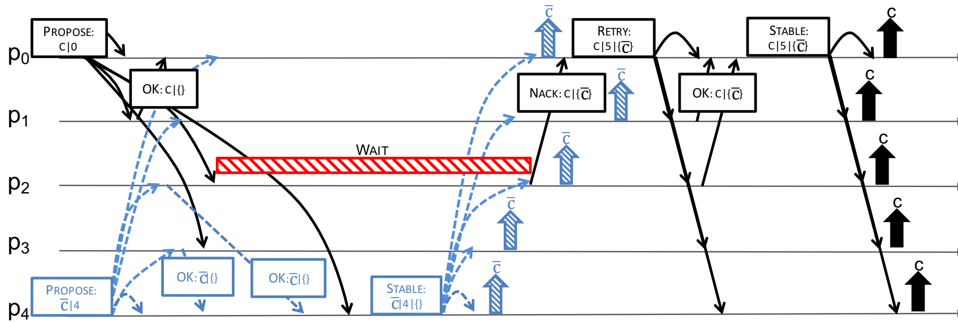
The rest of the overview addresses the following questions: what does a node do if it observes out of order timestamps? (Section 3.2.2) And, how does a command's leader behave if one of the nodes in the replying quorum rejects a proposed timestamp? (Section 3.2.2)

Out of Order Timestamps

Let us now consider the scenario in Figure 3.2(a), where, unlike the one in Figure 3.1, 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



(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.

Figure 3.2: 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 $\mathcal{T} = 0 < \bar{\mathcal{T}} = 4$.

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 (red bar labeled with WAIT along N_2's timeline in Figure 3.2(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 any deadlock in the execution, since only commands with a lower timestamp, e.g., c, wait for the final decision of conflicting commands with a higher timestamp, e.g., $\bar{c}$.

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 3.2(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. 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 3.2.2. Therefore, in Figure 3.2(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 4.3 and Chapter 6 for further details). 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.

Chapter 4

Protocol Details

This chapter describes Caesar, by detailing the required data structures in Section 4.1, the procedure for a fast decision in Section 4.2, the procedure for a slow decision in Section 4.3, and the behavior of the protocol in case of failures in Section 4.5. The case when a leader is not able to contact a fast quorum of nodes during the execution of the *fast proposal phase* for a command despite the availability of more than f nodes, is also explained. 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 of the protocol is in Section 4.4.

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*.

As in most multi-leader consensus protocols, a command c is proposed to 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*, by finalizing 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 the execution of the final *stable phase*. This results in a *slow decision*, with an overall latency of four communication delays.

The main pseudocode for Caesar is presented in Figure 4.2. 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 4.1 and 4.3, respectively.

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 (e.g., natural numbers), 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/receives a command with timestamp $\mathcal{T}$, it updates its own $\mathcal{TS}_i$ with a value that is greater than $\mathcal{T}$. 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 can be 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). Obviously, we can also define the total order relation on those values 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 history of events handled 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 timestamp of c for this event; $\mathcal{P}red$ 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-pending*, *slow-pending*, *accepted*, *rejected*, *stable*$\}$; $\mathcal{B}$ is the ballot number associated with this event, and it has values in $\mathbb{N}$; *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 for each command c. 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 $\mathcal{H}_i$;
$\mathcal{H}_i$.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 the empty sequence.

$\underline{\mathcal{B}allots_i}$

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.

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 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 $\mathcal{B}$.

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 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 tuple for c in its local history $\mathcal{H}_j$ by marking that as *fast-pending* with $\mathcal{T}ime$ and $\mathcal{P}red_j$ (line P14), and calls the function WAIT (line P15) to check the wait condition, as introduced in Section 3.2.2. 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 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 5–10 of Figure 4.1) 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 timestamp proposed by the leader, 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 lesser than c's timestamp, with the following exception (see lines 1–3 of Figure 4.1): 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-pending* in $\mathcal{H}_j$ in order to be included in $\mathcal{P}red_j$. In fact, $\mathcal{W}hitelist$ contains the commands that should be considered as predecessors of c according to perception of the node that is executing a recovery procedure for c (see Section 4.5).

In case of a *fast decision* (see *FastDecision* transition in Figure 4.2), 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 [19, 11], a fast decision in Caesar is guaranteed in case a fast quorum of nodes confirms the timestamp for a command, although those nodes can reply with non equal predecessors sets. The correctness proof of Caesar (see Chapter 6) shows 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 $\mathcal{T}ime$ after that 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 20–21 of Figure 4.1), p_j can decide c by triggering Decide(c) (lines S5–S7). This is correct because, as we will prove in Chapter 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

```
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:
5: function Boolean Wait(c, Time)
6:     wait until ∀⟨c̄, T̄, Pred‾, −, −, −⟩ ∈ H_j,
             (c̄ ∼ c ∧ Time < T̄ ∧ c ∉ Pred‾ ⇒
                ∃⟨c̄, T̄, Pred‾, accepted/stable, −, −⟩ ∈ H_j)
7:     if ∃⟨c̄, T̄, Pred‾, accepted/stable, −, −⟩ ∈ H_j :
                          c̄ ∼ c ∧ Time < T̄ ∧ c ∉ Pred‾ then
8:         return NACK
9:     else
10:        return OK
11:
12: function BreakLoop(c)
13:     ⟨c, T, Pred, stable, B, ⊥⟩ ← H_j.Get(c)
14:     for all c̄ ∈ Pred : ⟨c̄, T̄, Pred‾, stable, B‾, ⊥⟩ ∈ H_j ∧ T̄ < T do
15:         H_j.Update(c̄, T̄, Pred‾ \ {c}, stable, B‾, ⊥)
16:     for all c̄ ∈ Pred : ⟨c̄, T̄, Pred‾, stable, B‾, ⊥⟩ ∈ H_j ∧ T̄ > T do
17:         Pred ← Pred \ {c̄}
18:     H_j.Update(c, T, Pred, stable, B, ⊥)
19:
20: function Boolean Deliverable(c)
21:     return (c ∪ H_j.GetPredecessors(c)) ⊆ Decided_j
```

Figure 4.1: Auxiliary functions - node p_j

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 $\overline{\mathcal{T}ime} < \mathcal{T}ime$, and 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 12–18 of Figure 4.1). To do that, Caesar adopts a simple and easy to parallelize procedure, unlike the graph processing adopted by EPaxos that requires a module to identify the strongly connected components of the commands' dependency graphs. In Caesar, 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.

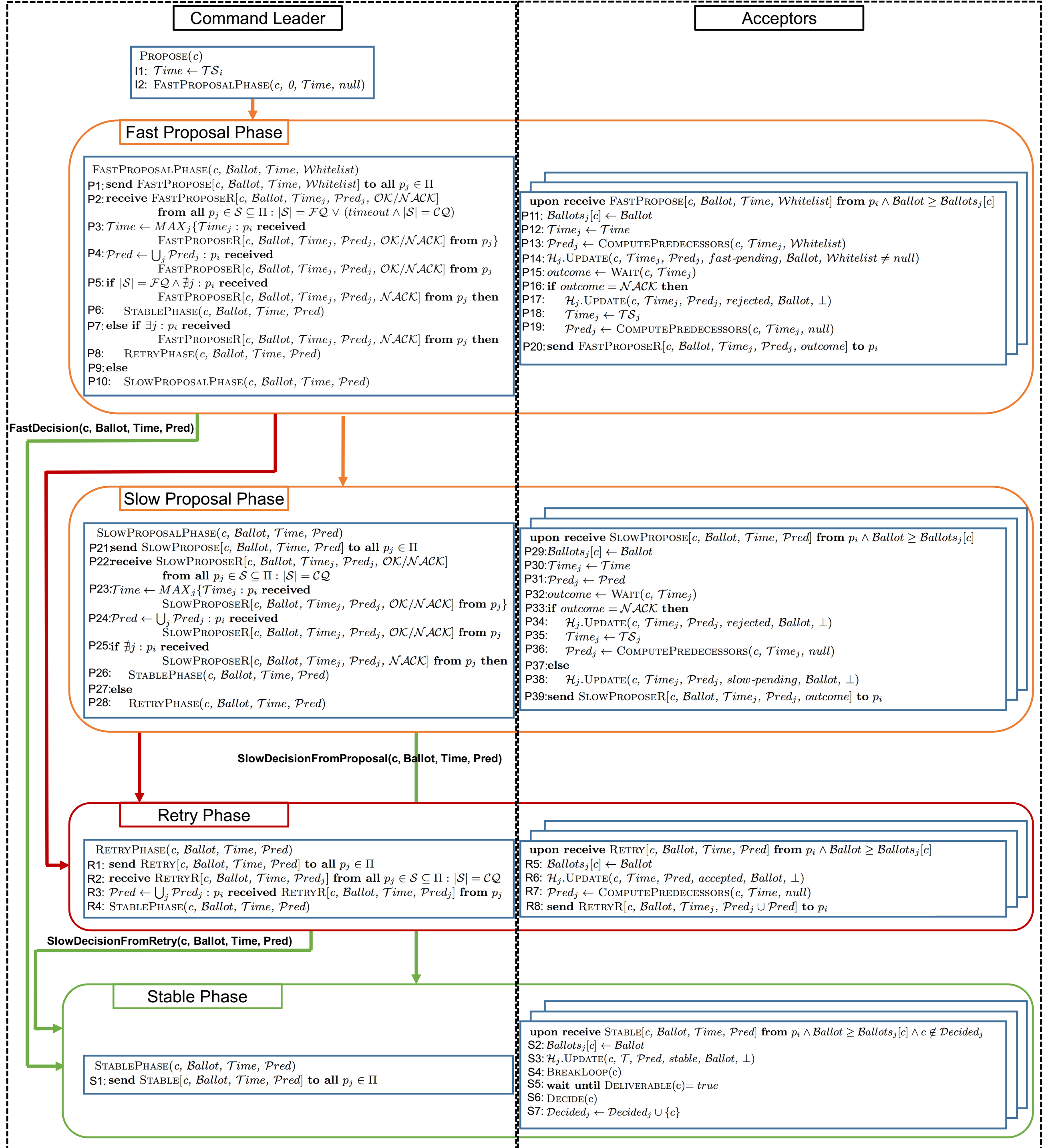


Figure 4.2: Pseudocode of CAESAR. The left part is executed by the command c's leader p_i, and the right part can be executed by any acceptor p_j (including p_i).

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, 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); the leader is only able to collect a classic 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 4.4.

The *slow decision* is implemented as follows. After a client proposes c and c's command leader executes the *fast proposal phase* for c (lines I1–I2, P1–P4, and P11–P20), c has to undergo an additional *retry phase* before it is broadcast in the *stable phase*, since the leader received a rejection of the proposed timestamp among the fast quorum of $\mathcal{FQ}$ FASTPROPOSER messages (lines P7–P8, and R1–R8).

Retry phase

The purpose of this phase is to guarantee that the outcome of the previous *proposal phase* for a command c is accepted by a quorum of $\mathcal{CQ}$ nodes before moving to the decision of c in the *stable phase*, as described in Section 4.2. 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 in case of f failures, there always exists a correct node that confirmed $\mathcal{T}ime$ in this phase.

It is important to notice that, as 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 we explained in Section 3.2.2, 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 Chapter 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, $\mathcal{T}ime$, and $\mathcal{P}red$, it only updates the tuple for c in its local history $\mathcal{H}_j$ 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).

4.4 Unavailability of Fast Quorums

In CAESAR, as in other fast consensus implementations [11], 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).

The reason behind this design choice is the following: intuitively, if a command leader did not collect a fast quorum of $\mathcal{OK}$ replies, it cannot take a decision in two communication delays by directly executing the *stable phase*, due to the lower bound on fast consensus defined in [25]. Therefore, after having collected a quorum of $\mathcal{CQ}$ replies in the *fast proposal phase*, and if none of them has rejected the proposed timestamp, the leader is required to execute an additional communication phase, i.e., the *slow proposal phase*, by contacting a quorum of $\mathcal{CQ}$ acceptors, in order to ensure that even after f failures, there will always be a correct node having information about the proposed timestamp.

Note that, the role played by the *slow proposal phase* is similar to the one played by the *retry phase*, with the difference that, unlike the case of the retry, an acceptor can still reject a proposed timestamp for c in the *slow proposal phase*. For the sake of clarity, we remind to Sections 4.5 and Chapter 6 for more details.

The execution of the *slow proposal phase* resembles the execution of the *fast proposal phase*, with the following two exceptions: obviously, the predecessors set $\mathcal{P}red$, which has been computed in the *fast proposal phase*, has to be broadcast as part of a SLOWPROPOSE message in the *slow proposal phase* (lines P21 and P31); a node p_j that receives a proposal of a timestamp $\mathcal{T}$ and a set $\mathcal{P}red$ for c in the *slow proposal phase*, marks c as *slow-pending* in

$\mathcal{H}_j$ if the WAIT function does not reject $\mathcal{T}$ (lines P32, P37–P38).

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 4.2, p_k would decide c only after having received a STABLE message from p_i.

For this reason, CAESAR also includes an explicit recovery procedure, whose pseudocode is shown in Figure 4.3, 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 finalize 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 of 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 4.2).

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*; each indicate: 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 4.2). *iv)* If there exists a tuple with status *slow-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 $fast$-*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 4.2, and lines 1–3 of Figure 4.1).

```
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:
30: upon receive Recovery[c, Ballot] from p_k ∧ Ballot > Ballots_j[c]
31:     Ballots_j[c] ← Ballot
32:     if H_j.Contains(c) then
33:         send RecoveryR[c, Ballots_j[c], H_j.GetInfo(c)] to p_k
34:     else
35:         send RecoveryR[c, Ballots_j[c], NOP] to p_k
```

Figure 4.3: Recovery phase executed by node p_k. Node p_j is a receiver of the Recovery message.

Chapter 5

Implementation and Evaluation

This chapter presents the implementation details of CAESAR followed by a comprehensive evaluation study. Section 5.1 explains how CAESAR was implemented along with an overview of optimizations. Section 5.2 presents a detailed evaluation of CAESAR and contrasts with other state-of-the-art implementations, under both non-faulty and faulty scenarios.

5.1 Implementation

CAESAR was implemented in Java 8 by adopting and modifying JPaxos [26], a Java library and runtime system for efficient state machine replication. Specifically, we used the network, messaging, and state machine abstraction layers of JPaxos and built the rest of the protocol on top of these abstractions. We particularly exploited the lambda functions feature of Java 8. In order to facilitate concurrency, CAESAR uses separate queues for handling different types of messages, and each of these queues is handled by a separate pool of threads. This optimization has been borrowed from the EPaxos implementation [11]. In addition, conflicting commands are tracked using a Red-Black tree data structure ordered by their timestamp, facilitating quick and efficient construction of the dependency set of commands. Moreover, there is a clear abstraction between the protocol and the benchmark implementations, making it easy to adopt existing or implement new applications on top of the protocol.

CAESAR was contrasted with four state-of-the-art consensus protocols: M^2Paxos, EPaxos, Multi-Paxos, and Mencius. The implementations of EPaxos, Multi-Paxos, and Mencius has been open-sourced by its authors on Github [27], while the implementation of M^2Paxos has been open-sourced by its authors on Bitbucket [28]. These available implementations are in the Go language. It must be noted that the Go implementations compile to native binary while the Java implementation of CAESAR runs on the Java Virtual Machine. Thus, a warmup phase was used before each of the experiments in order to kickstart the Java's

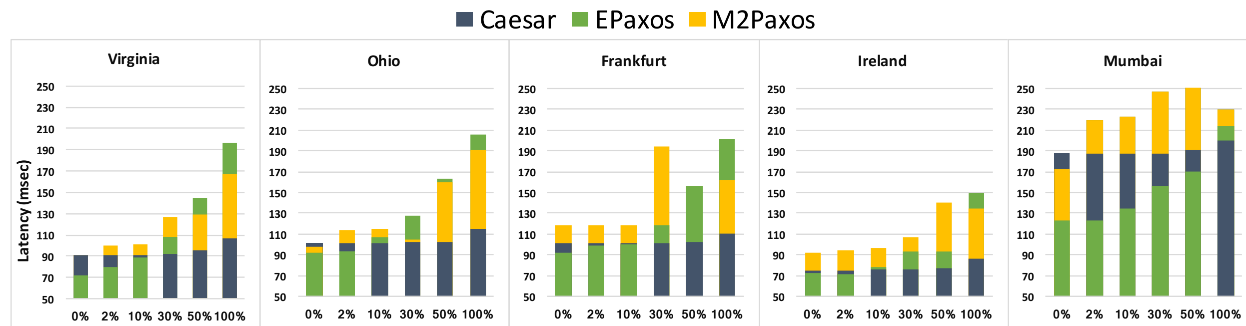


Figure 5.1: Average latency for ordering and processing commands by changing the percentage of conflicting commands. Batching disabled. Given a percentage value, bars of competitors are overlapped: e.g., in the case of 30% conflicting commands and the node in Virginia, latency values are 90 msec, 108 msec, and 127 msec, for Caesar, EPaxos, and M^2Paxos, respectively.

Hotspot JIT Compiler.

5.2 Evaluation

Caesar and its competitors were evaluated on Amazon EC2, using `m4.2xlarge` instances (8 vCPU and 32 GB Memory), running Ubuntu Linux 16.04. We ran a fully replicated Key-Value Store benchmark for evaluation. In this benchmark, clients issue commands to update a given key. We say two commands are conflicting if they access the same key. We chose this benchmark because it is representative of many realistic distributed applications widely in use today (e.g., the recently popular NoSQL databases) and it is already implemented and tested by all competitors. 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 and protocol statistics, 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 every competitor on five nodes spread around the world. Nodes are 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 fairly the same for all quorum-based competitors, however it is worth to recall that in a system with 5 nodes,

Table 5.1: Average round trip latencies (in ms) between sites

	Ohio	**Frankfurt**	**Ireland**	**Mumbai**
Virginia	11.263	90.376	80.693	186.22
Ohio		100.47	79.658	301.895
Frankfurt			21.474	112.363
Ireland				122.101

Caesar requires to contact one node more than other quorum-based competitors to reach a fast decision. Table 5.1 shows the measured round trip time (RTT) between different sites.

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.

5.2.1 Non-faulty Scenarios

In Figure 5.1, 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% of 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 than EPaxos because of the need of contacting one more node to reach consensus. However, this overhead is on average 18%.

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.

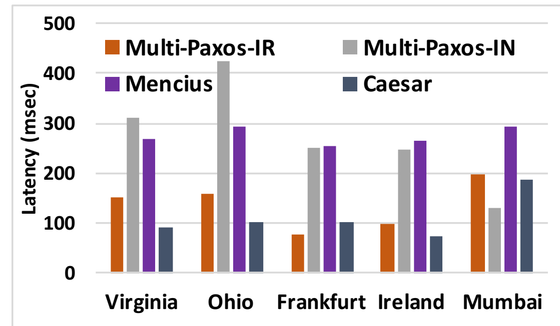


Figure 5.2: Average latency for ordering commands of Multi-Paxos (with a close and faraway leader), Mencius, and Caesar. Batching is disabled.

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.

Performance of Multi-Paxos and Mencius is reported in Figure 5.2 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, 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 5.1, Multi-Paxos-IR and Multi-Paxos-IN are, on average, 5% and 40% slower than Caesar 100%, respectively.

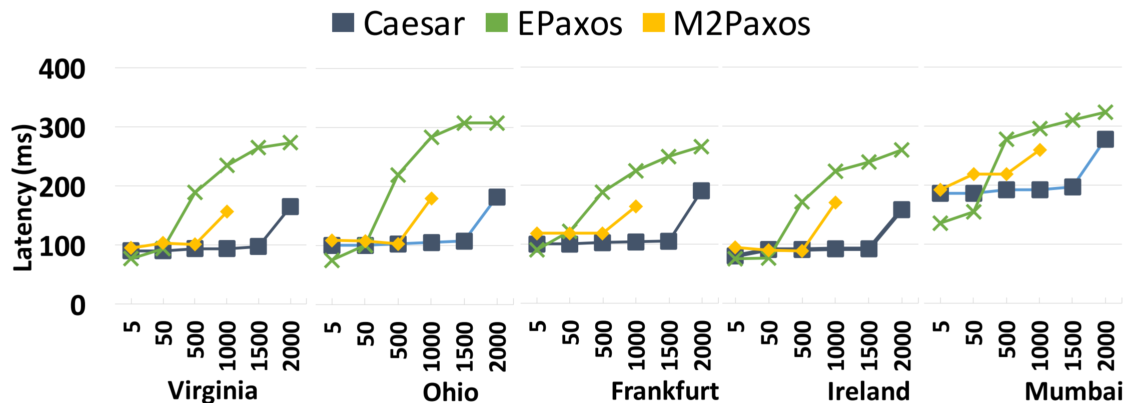


Figure 5.3: Latency per node while varying the number of connected clients (5 – 2000). Network messages are not batched.

Scalability of competitors is measured by loading the system with more clients. Figure 5.3 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.

Figure 5.4 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 is the reason behind 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 settings their CPU cycles needed to handle incoming messages are comparable.

In the bottom part of the plot, batching has been enabled. Mencius's implementation does not support batching thus we omitted it. Trend is similar to the one observed without having batching enabled. 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.

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 5.5, 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

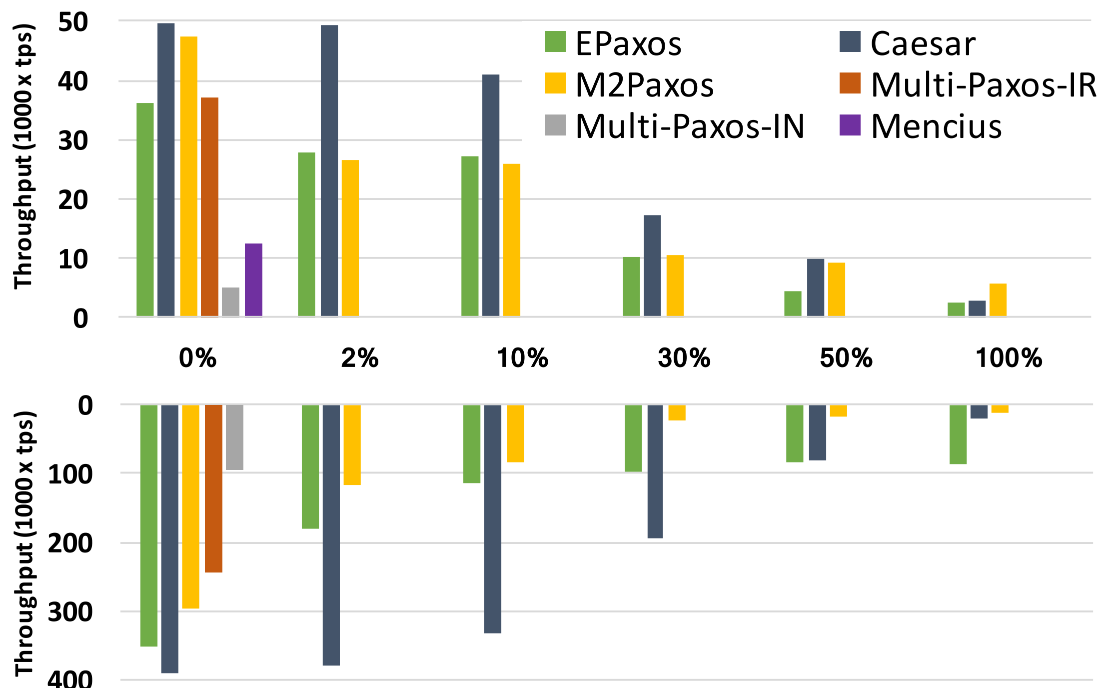


Figure 5.4: Throughput by varying the percentage of conflicting commands. In the top part of the plot batching is disabled, in the lower part it is enabled.

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 5.6, we report the internal statistics of Caesar. Figure 5.6(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 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 5.6(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; and given that faraway nodes are not aware of this advancement, they propose commands with a lower timestamp, which causes their conflicting commands to wait.

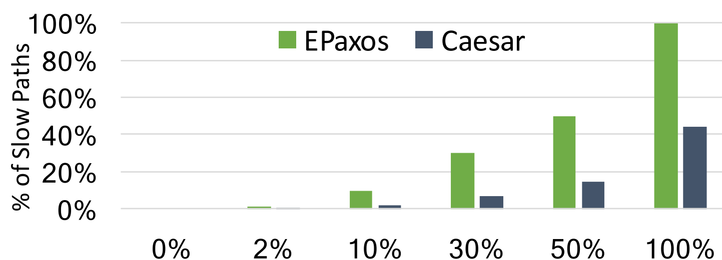
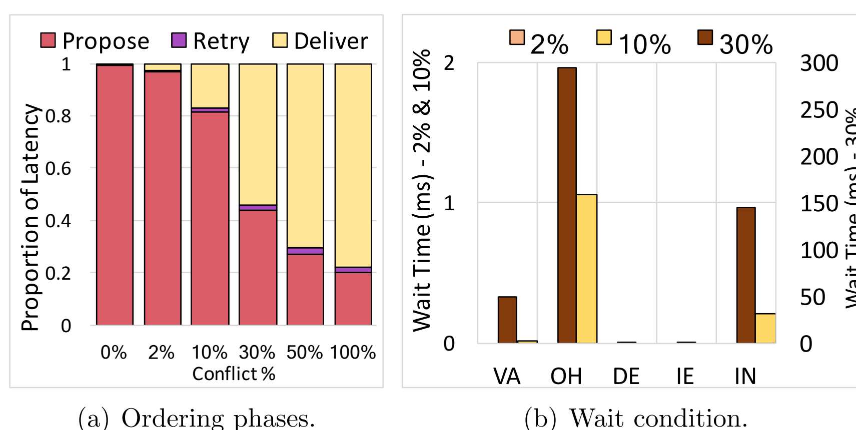


Figure 5.5: % of commands delivered using a slow decision by varying % of conflicting commands. Batching here is disabled.



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

Figure 5.6: Latency breakdown for Caesar.

5.2.2 Recovery

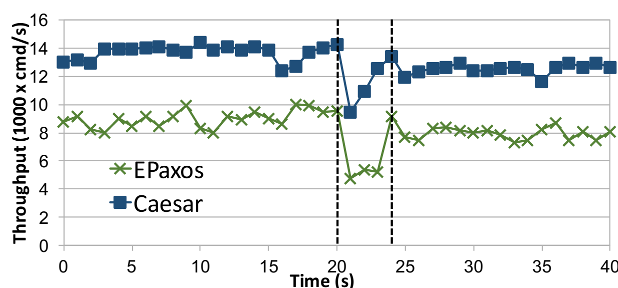


Figure 5.7: Throughput when one node fails.

In Figure 5.7, we report the throughput when one node crashes, to show that is 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.

Chapter 6

Correctness

This chapter provides the formal proof of correctness of CAESAR. Section 6.1 provides an overview of how CAESAR provides safety and liveness guarantees. Correctness is proved by showing that Theorems 1 and 2 hold in Section 6.3. In order to show that the theorems hold, Section 6.3 bases the proof of Lemmas 1–11. Section 6.2 introduces the preliminary terminologies used in Section 6.3.

6.1 Overview

CAESAR satisfies all the three safety requirements of a traditional consensus problem:

- *Nontriviality*: Any node only decides commands that were proposed. For any command c, DECIDE(c) is only called after c is *stable* and DELIVERABLE(c) returns *true* (S2–S7 in Figure 4.2). For this to happen, c should have been proposed using PROPOSE(c) (I1–I2 in Figure 4.2) and should have at least taken the *fast proposal phase*.
- *Stability*: The set of decided commands monotonically grows on each node.
- *Correctness*: The proof is provided in Section 6.3.

Liveness is guaranteed since any command is eventually assigned to a correct leader, which can eventually finalize the decision for it. Even under replica crashes, there is at least one node that has information about the commands owned by the crashed replica if at least the *fast proposal phase* was successful. Thus, the recovery algorithm in Section 4.5 ensures that a leader is eventually assigned (with the help of $\mathcal{B}allot$ numbers).

6.2 Terminology

The predicates are defined as follows:

- DECIDED[c,$\mathcal{T}$,$\mathcal{P}red$,$\mathcal{B}$] is equal to true whenever a node decides a command c with timestamp $\mathcal{T}$, predecessors set $\mathcal{P}red$, and ballot $\mathcal{B}$.

- FASTDECISION[c,$\mathcal{T}$,$\mathcal{P}red$,$\mathcal{B}$] is equal to true whenever a node decides a command c with timestamp $\mathcal{T}$, predecessors set $\mathcal{P}red$, and ballot $\mathcal{B}$ in a fast decision. This means that the command is decided in a *stable phase* after a transition from a fast proposal phase (see transition FASTDECISION in the pseudocode of Figure 4.2).

- SLOWDECISIONFROMRETRY[c,$\mathcal{T}$,$\mathcal{P}red$,$\mathcal{B}$] is equal to true whenever a node decides a command c with timestamp $\mathcal{T}$, predecessors set $\mathcal{P}red$, and ballot $\mathcal{B}$ in a slow decision after the execution of a *retry phase*. This means that the command is decided in a *stable phase* after a transition from a *retry phase* (see transition SLOWDECISIONFROMRETRY in the pseudocode of Figure 4.2).

- SLOWDECISIONFROMPROPOSAL[c,$\mathcal{T}$,$\mathcal{P}red$,$\mathcal{B}$] is equal to true whenever a node decides a command c with timestamp $\mathcal{T}$, predecessors set $\mathcal{P}red$, and ballot $\mathcal{B}$ in a slow decision after the execution of a *slow proposal phase*. This means that the command is decided in a *stable phase* after a transition from a *slow proposal phase* (see transition SLOWDECISIONFROMPROPOSAL in the pseudocode of Figure 4.2).

- SLOWDECISIONFROMRETRYFP[c,$\mathcal{T}$,$\mathcal{P}red$,$\mathcal{B}$] is equal to true whenever a node decides a command c with timestamp $\mathcal{T}$, predecessors set $\mathcal{P}red$, and ballot $\mathcal{B}$ in a slow decision after the execution of a *retry phase*, which is executed due to a rejection in the *fast proposal phase*. This means that the command is decided in a *stable phase* after a transition from *fast proposal phase* through a *retry phase*.

- SLOWDECISIONFROMRETRYSP[c,$\mathcal{T}$,$\mathcal{P}red$,$\mathcal{B}$] is equal to true whenever a node decides a command c with timestamp $\mathcal{T}$, predecessors set $\mathcal{P}red$, and ballot $\mathcal{B}$ in a slow decision after the execution of a *retry phase*, which is executed due to a rejection in the *slow proposal phase*. This means that the command is decided in a *stable phase* after a transition from *slow proposal phase* through a *retry phase*.

6.3 Proof on Consistency

Lemma 1. $\forall c, \bar{c}$, (FASTDECISION[$c, \mathcal{T}, \mathcal{P}red, \mathcal{B}$] $\wedge$
(SLOWDECISIONFROMRETRY[$\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \overline{\mathcal{B}}$]$\vee$SLOWDECISIONFROMPROPOSAL[$\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \overline{\mathcal{B}}$])
$\wedge\ \bar{\mathcal{T}} < \mathcal{T} \wedge c \sim \bar{c}$) $\Rightarrow \bar{c} \in \mathcal{P}red$.

Proof. $\forall\ c, \mathcal{T}, \mathcal{P}red, \mathcal{B} : \textsc{FastDecision}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \Rightarrow$
$\forall \bar{c} \notin \mathcal{P}red : \bar{c} \sim c, \exists \mathcal{FQ} \in FastQuorums : \forall p_j \in \mathcal{FQ}, \exists \langle c, \mathcal{T}, \mathcal{P}red, fast\text{-}pending/stable, \mathcal{B}, - \rangle \in H_j \wedge \bar{c} \notin \mathcal{P}red_j \wedge (\exists \langle \bar{c}, \bar{\mathcal{T}}, -, -, -, - \rangle \in H_j \wedge \bar{\mathcal{T}} < \mathcal{T})$

$$\Downarrow$$

$$(\exists \langle \bar{c}, \bar{\mathcal{T}}, -, fast\text{-}pending, -, - \rangle \in H_j \vee \textsc{Wait}(\bar{c}, \bar{\mathcal{T}}, p_j) \vee \exists \langle \bar{c}, \bar{\mathcal{T}}, -, rejected, -, - \rangle \in H_j)$$

$$\wedge$$

$$\nexists \langle \bar{c}, \bar{\mathcal{T}}, -, accepted, -, - \rangle \in H_j$$

$$\wedge$$

$$\nexists \langle \bar{c}, \bar{\mathcal{T}}, -, stable, -, - \rangle \in H_j$$

$$\wedge$$

$$\nexists \langle \bar{c}, \bar{\mathcal{T}}, -, slow\text{-}pending, -, - \rangle \in H_j$$

$\bar{c}$ is not decided slow at $\bar{\mathcal{T}}$ because

$\textsc{SlowDecision}[\bar{c}, \bar{\mathcal{T}}, -, \bar{\mathcal{B}}] \Rightarrow$
$\forall \mathcal{CQ} \in ClassicQuorums, \exists p_j \in \mathcal{CQ} : \exists \langle \bar{c}, \bar{\mathcal{T}}, -, slow\text{-}pending/accepted/stable, \bar{\mathcal{B}}_2 \geq \bar{\mathcal{B}} \rangle \in H_j$

And it won't be decided at $\bar{\mathcal{T}}$ because as we saw above:

$\forall c, \mathcal{T}, \mathcal{P}red, \mathcal{B} : \textsc{FastDecision}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \Rightarrow$
$\forall \bar{c} \notin \mathcal{P}red : \bar{c} \sim c, \exists \mathcal{CQ} \in ClassicQuorums : \forall p_j \in \mathcal{CQ}, (\exists \langle \bar{c}, \bar{\mathcal{T}}, -, -, -, - \rangle \in H_j \wedge \bar{\mathcal{T}} < \mathcal{T})$

$$\Downarrow$$

$$(\exists \langle \bar{c}, \bar{\mathcal{T}}, -, fast\text{-}pending, -, -, - \rangle \in H_j \vee (\textsc{Wait}(\bar{c}, \bar{\mathcal{T}}, p_j) \vee \exists \langle \bar{c}, \bar{\mathcal{T}}, -, rejected, -, - \rangle \in H_j))$$

$$\wedge$$

$$\nexists \langle \bar{c}, \bar{\mathcal{T}}, -, slow\text{-}pending, -, -, - \rangle \in H_j$$

$$\wedge$$

$$\nexists \langle \bar{c}, \bar{\mathcal{T}}, -, accepted, -, -, - \rangle \in H_j$$

$$\wedge$$

$$\nexists \langle \bar{c}, \bar{\mathcal{T}}, -, stable, -, -, - \rangle \in H_j$$

□

Lemma 2. $\forall c, \bar{c}, (\textsc{FastDecision}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge \textsc{FastDecision}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \overline{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge c \sim \bar{c} \Rightarrow \bar{c} \in \mathcal{P}red)$.

Proof. $\textsc{FastDecision}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge \textsc{FastDecision}[\bar{c}, \bar{\mathcal{T}}, \bar{\mathcal{P}}red, \bar{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge \bar{c} \notin \mathcal{P}red$
$\exists \mathcal{FQ} \in FastQuorums : \forall p_j \in \mathcal{FQ}, \exists \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red_j}, fast\text{-}pending/stable, \bar{\mathcal{B}}, - \rangle \in H_j \wedge$
$\neg \textsc{Wait}(\bar{c}, \bar{\mathcal{T}}, p_j) \wedge \nexists \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, rejected/accepted/slow\text{-}pending, \bar{\mathcal{B}}, - \rangle \in H_j$

$\Rightarrow \exists \mathcal{FQ}_1, \mathcal{FQ}_2 \in FastQuorums : \forall p_j \in \mathcal{FQ}_1 \cap \mathcal{FQ}_2 : \exists \langle c, \mathcal{T}, \mathcal{P}red_j, fast\text{-}pending/stable, \mathcal{B}, - \rangle \in H_j \wedge \bar{c} \notin \mathcal{P}red_j \wedge \exists \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red_j}, fast\text{-}pending/stable, \bar{\mathcal{B}}, - \rangle \in H_j \wedge \neg \textsc{Wait}(\bar{c}, \bar{\mathcal{T}}, p_j)$

only if $\bar{c}$ was not in some whitelist for c due to the intersections of $FastQuorums$

$\exists \mathcal{B}' \leq \mathcal{B}, \exists \mathcal{CQ} \in ClassicQuorums : \forall p_k \in \mathcal{CQ} : \exists \langle c, \mathcal{T}, \mathcal{P}red_k, fast\text{-}pending, \mathcal{B}', \perp \rangle \in H_k \wedge \ \exists \textsc{Maj} \subseteq \mathcal{CQ} : |\textsc{Maj}| = \lfloor \frac{|\mathcal{CQ}|}{2} \rfloor + 1 \wedge$
$\forall p_h \in \text{MAJ}, (\exists \langle c, \mathcal{T}, \mathcal{P}red_h, fast\text{-}pending, \mathcal{B}', \perp \rangle \in H_h \wedge \bar{c} \notin \mathcal{P}red_h \wedge (\nexists \langle \bar{c}, \bar{\mathcal{T}}, -, -, -, - \rangle \in H_h \vee \textsc{Wait}(\bar{c}, \bar{\mathcal{T}}, p_h) \vee \exists \langle \bar{c}, \bar{\mathcal{T}}, -, rejected, -, - \rangle \in H_h)$

$$\forall \mathcal{FQ} \in FastQuorums, \forall \mathcal{CQ} \in ClassicQuorums, |\mathcal{FQ} \cap \mathcal{CQ}| \geq \lfloor \frac{|\mathcal{CQ}|}{2} \rfloor + 1$$

OR

$$\forall \mathcal{FQ}_1, \mathcal{FQ}_2 \in FastQuorums, \forall \mathcal{CQ} \in ClassicQuorums, \mathcal{FQ}_1 \cap \mathcal{FQ}_2 \cap \mathcal{CQ} \neq \emptyset$$

$$\Downarrow$$

$$\forall \mathcal{FQ} \in FastQuorums, \exists p_s \in \mathcal{FQ} : \nexists \langle \bar{c}, \bar{\mathcal{T}}, -, -, -, - \rangle \in H_h \vee \textsc{Wait}(\bar{c}, \bar{\mathcal{T}}, p_s) \vee \exists \langle \bar{c}, \bar{\mathcal{T}}, -, rejected, -, - \rangle \in H_h$$

$$\Rightarrow \nexists \overline{\mathcal{P}red}, \bar{\mathcal{B}} : \textsc{FastDecision}(\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}})$$

□

Lemma 3. $\forall c, \bar{c}, (\textsc{SlowDecisionFromProposal}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge \textsc{FastDecision}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \overline{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge c \sim \bar{c} \Rightarrow \bar{c} \in \mathcal{P}red)$.

Proof. Assume that,

$\textsc{SlowDecisionFromProposal}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge \textsc{FastDecision}[\bar{c}, \bar{\mathcal{T}}, \bar{\mathcal{P}}red, \bar{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge \bar{c} \notin \mathcal{P}red \wedge c \sim \bar{c}$

$\textsc{SlowDecisionFromProposal}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \Rightarrow$

$\exists \mathcal{CQ} \in ClassicQuorums, \forall p_j : \exists \langle c, \mathcal{T}, \mathcal{P}red, slow\text{-}pending/stable, \mathcal{B}, \perp \rangle \in H_j \wedge \bar{c} \notin \mathcal{P}red_j$

$\textsc{FastDecision}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}}] \Rightarrow$

$\exists \mathcal{FQ} \in FastQuorums : \forall p_j \in \mathcal{FQ} : \exists \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \textit{fast-pending/stable}, \bar{\mathcal{B}}, - \rangle \in H_j$

$\wedge \neg \textsc{Wait}(\bar{c}, \bar{T}, p_j) \wedge \nexists \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, rejected, \bar{\mathcal{B}}, - \rangle \in H_j \wedge \nexists \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red_j}, accepted, \bar{\mathcal{B}}, - \rangle \in H_j$

$\wedge \nexists \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red_j}, \textit{slow-pending}, \bar{\mathcal{B}}, - \rangle \in H_j$

$\textsc{SlowDecision}(c, \mathcal{T}, \mathcal{P}red, \mathcal{B}) \Rightarrow \exists \mathcal{B}' \leq \mathcal{B} : \exists \mathcal{CQ} \in ClassicQuorums, \forall p_k \in \mathcal{CQ} :$

(

$\exists \langle c, \mathcal{T}, \mathcal{P}red_k, \textit{fast-pending}, \mathcal{B}', - \rangle \in H_k \wedge \bar{c} \notin \mathcal{P}red_k$

$\wedge \nexists \langle \bar{c}, \bar{\mathcal{T}}, -, -, -, - \rangle \in H_k \vee \textsc{Wait}(\bar{c}, \bar{\mathcal{T}}, p_k) \vee \exists \langle \bar{c}, \bar{\mathcal{T}}, -, rejected, -, - \rangle \in H_k$

$\vee \exists \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red_k}, \textit{fast-pending}, \bar{\mathcal{B}}, - \rangle$

$\wedge \exists \mathcal{B}'' \leq \mathcal{B}', \exists \mathcal{CQ} \in ClassicQuorums : \forall p_h \in \mathcal{CQ}'' : \exists \langle c, \mathcal{T}, \mathcal{P}red_h, \textit{fast-pending}, \mathcal{B}'', \perp \rangle \in H_h$

$\wedge \exists \textsc{Maj} \subseteq \mathcal{CQ}'' : |\textsc{Maj}| = \lfloor \frac{|\mathcal{CQ}''|}{2} \rfloor + 1 \ \wedge \forall p_s \in \textsc{Maj}$

(

$\exists \langle c, \mathcal{T}, \mathcal{P}red_s, \textit{fast-pending}, \mathcal{B}'', \perp \rangle \in H_s \wedge \bar{c} \notin \mathcal{P}red_s$

$\wedge (\nexists \langle \bar{c}, \bar{\mathcal{T}}, -, -, -, - \rangle \in H_s \vee \textsc{Wait}(\bar{c}, \bar{\mathcal{T}}, p_h) \vee \exists \langle \bar{c}, \bar{\mathcal{T}}, -, rejected, -, - \rangle \in H_s)$

)

)

$$\Downarrow$$

$$\text{Since } \forall \mathcal{CQ} \in ClassicQuorum, \forall \mathcal{FQ} \in FastQuorums, \mathcal{CQ} \cap \mathcal{FQ} \neq \emptyset \wedge \forall \mathcal{FQ}_1, \mathcal{FQ}_2 \in FastQuorums, \forall \mathcal{CQ} \in ClassicQuorums \vee \mathcal{FQ}_1 \cap (\mathcal{FQ}_2 \cap \mathcal{CQ}) \neq \emptyset \wedge |\mathcal{FQ}_2 \cap \mathcal{CQ}| \geq \lfloor \frac{|\mathcal{CQ}|}{2} \rfloor + 1$$

$$\Downarrow$$

$$\forall \mathcal{FQ} \in FastQuorums, \exists p_i \in \mathcal{FQ}, \nexists \langle \bar{c}, \bar{\mathcal{T}}, -, -, -, -, \rangle \in H_i \vee \textsc{Wait}(\bar{c}, \bar{\mathcal{T}}, p_i) \vee \exists \langle \bar{c}, \bar{\mathcal{T}}, -, rejected, -, - \rangle \in H_i$$

$$\Downarrow$$

$$\neg \textsc{FastDecision}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}}]$$

This is a Contradiction.

□

Lemma 4. $\forall c, \bar{c}, (\textsc{SlowDecisionFromRetry}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge \textsc{FastDecision}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \overline{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge c \sim \bar{c} \Rightarrow \bar{c} \in \mathcal{P}red)$.

Proof. Assume that

$\textsc{SlowDecisionFromRetry}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \ \wedge\ \textsc{FastDecision}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge \bar{c} \notin \mathcal{P}red$

Thus,

$\textsc{SlowDecisionFromRetry}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \Rightarrow$

$\exists \mathcal{CQ} \in ClassicQuorums, \forall p_i \in \mathcal{CQ} : \exists \langle c, \mathcal{T}, \mathcal{P}red_i, accepted, \mathcal{B}, \perp \rangle \in H_i \ \wedge$

$(\nexists \langle \bar{c}, \bar{\mathcal{T}}, -, -, -, - \rangle \in H_i \vee \exists \langle \bar{c}, \bar{\mathcal{T}}, -, rejected, -, - \rangle \in H_i)$

$\Rightarrow \neg \textsc{FastDecision}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \mathcal{B}]$

This is a contradiction.

□

Lemma 5. $\forall c, \bar{c}, (\textsc{SlowDecisionFromProposal}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge \textsc{SlowDecisionFromProposal}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \overline{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge c \sim \bar{c} \Rightarrow \bar{c} \in \mathcal{P}red)$

Proof. $\textsc{SlowDecisionFromProposal}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge \textsc{SlowDecisionFromProposal}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge \bar{c} \notin \mathcal{P}red \wedge \bar{c} \sim c$

$\textsc{SlowDecisionFromProposal}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \Rightarrow$

$\exists \mathcal{B}' \leq \mathcal{B} : \exists \mathcal{CQ} \in ClassicQuorums : \forall p_j \in \mathcal{CQ}, \exists \langle c, \mathcal{T}, \mathcal{P}red, fast\text{-}pending, \mathcal{B}', - \rangle \in H_j \ \wedge\ \bar{c} \notin \mathcal{P}red_j$

$\textsc{SlowDecisionFromProposal}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}}] \Rightarrow$

$\exists \mathcal{B}'' \leq \bar{\mathcal{B}} : \exists \mathcal{CQ} \in ClassicQuorums : \forall p_k \in \mathcal{CQ}, \exists \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, fast\text{-}pending, \mathcal{B}'', - \rangle \in H_k \Rightarrow$

$\bar{c} \notin \mathcal{P}red \Rightarrow \exists \mathcal{B}' \leq \mathcal{B}, \exists \mathcal{CQ} \in ClassicQuorums : \forall p_j \in \mathcal{CQ}, \exists \langle c, \mathcal{T}, \mathcal{P}red_j, fast\text{-}pending, \mathcal{B}', - \rangle \in H_j \wedge \bar{c} \notin \mathcal{P}red_j \ \wedge$

(

$\exists \langle \bar{c}, \bar{\mathcal{T}}, -, -, -, - \rangle \in H_j \Rightarrow (\exists \langle \bar{c}, \bar{\mathcal{T}}, -, fast\text{-}pending, -, - \rangle \in H_j \vee \textsc{Wait}(\bar{c}, \bar{\mathcal{T}}, p_j))$

$\wedge \nexists \langle \bar{c} \bar{\mathcal{T}}, -, slow\text{-}pending, -, - \rangle \in H_j$

)

This is a contradiction if $\bar{c}$ is decided in a slow propose phase.

$\textsc{SlowDecisionFromProposal}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}}] \Rightarrow$

$\forall \mathcal{CQ} \in ClassicQuorums\ \exists p_k \in \mathcal{CQ} : \exists \langle \bar{c}, \bar{\mathcal{T}}, \mathcal{P}red_k, slow\text{-}pending, \bar{\mathcal{B}}, - \rangle \in H_k$

$\wedge \neg \textsc{Wait}(\bar{c}, \bar{\mathcal{T}}, p_k) \wedge \nexists \langle \bar{c}, \bar{\mathcal{T}}, -, rejected, \bar{\mathcal{B}}, - \rangle \in H_k$

$\square$

Lemma 6. $\forall c, \bar{c}, (\textsc{SlowDecisionFromProposal}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge$
$\textsc{SlowDecisionFromRetryFP}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \overline{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge c \sim \bar{c} \Rightarrow \bar{c} \in \mathcal{P}red)$.

Proof. Assume that,

$\textsc{SlowDecisionFromProposal}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge$
$\textsc{SlowDecisionFromRetryFP}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \overline{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge \bar{c} \notin \mathcal{P}red \wedge \bar{c} \sim c$

$\textsc{SlowDecisionFromProposal}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}} \Rightarrow$

$\exists \mathcal{CQ} \in ClassicQuorums : \forall p_j \in \mathcal{CQ}, \exists \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red_j}, accepted, \bar{\mathcal{B}}, - \rangle \in H_j \wedge \exists \mathcal{B}'' \leq \bar{\mathcal{B}},$

$\mathcal{T}_2 < \bar{\mathcal{T}} : \exists \mathcal{FQ} \in FastQuorums : \forall p_k \in \mathcal{FQ} : \exists \langle \bar{c}, \mathcal{T}_2, -, fast\text{-}pending/rejected, -, - \rangle \in H_k$

$\textsc{SlowDecisionFromRetryFP}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \Rightarrow$

$\exists \mathcal{B}' \leq \mathcal{B} : \exists \mathcal{CQ} \in ClassicQuorums, \forall p_k \in \mathcal{CQ}, \exists \langle c, \mathcal{T}, \mathcal{P}red_k, fast\text{-}prending, \mathcal{B}', - \rangle \in H_k \wedge$
$\bar{c} \notin \mathcal{P}red_k \wedge$

(

$\exists \langle \bar{c}, \mathcal{T}_3 < \mathcal{T}, -, -, -, - \rangle \in H_k \Rightarrow \textsc{Wait}(\bar{c}, \mathcal{T}_3, p_k) \vee \exists \langle \bar{c}, \mathcal{T}_3 < \mathcal{T}, -, rejected, -, - \rangle \in H_k \vee$

(

$\exists \langle \bar{c}, \mathcal{T}_3 < \mathcal{T}, -, fast\text{-}pending, \bar{\mathcal{B}}, - \rangle \wedge$

$\exists \mathcal{B}'' \leq \mathcal{B}', \exists \mathcal{CQ}'' \in ClassicQuorums : \forall p_k \in \mathcal{CQ}'', \exists \langle c, \mathcal{T}, \mathcal{P}red_h, fast\text{-}pending, \mathcal{B}'', \perp \rangle \in H_h$

$\wedge\ \exists \mathcal{FQ} \in FastQuorums : \forall p_s \in \mathcal{FQ} \cap \mathcal{CQ}'', \exists \langle c, \mathcal{T}, \mathcal{P}red_s, fast\text{-}pending, \mathcal{B}'', \perp \rangle \in H_s$

$\wedge\ \bar{c} \notin \mathcal{P}red_s \wedge (\exists \langle \bar{c}, \mathcal{T}_3 < \mathcal{T}, -, -, -, -, \rangle \in H_s \Rightarrow \textsc{Wait}(\bar{c}, \mathcal{T}_3, p_s) \vee \exists \langle \bar{c}, \mathcal{T}_3 < \mathcal{T}, -, rejected \rangle)$

)

)

Since $\mathcal{FQ} \cap \mathcal{CQ} \neq \emptyset$ and $\mathcal{FQ}_1 \cap \mathcal{FQ}_2 \cap \mathcal{CQ} \neq \emptyset$

$\forall \mathcal{FQ} \in FastQuorums, \exists p_i \in \mathcal{FQ} : \exists \langle \bar{c}, \mathcal{T}_3 < \mathcal{T}, -, -, -, - \rangle \in H_i \Rightarrow \textsc{Wait}(\bar{c}, \mathcal{T}, p_i) \vee$

$\exists\langle \bar{c}, \mathcal{T}_3, -, rejected, -, -\rangle \in H_i \Rightarrow$

$\nexists \mathcal{CQ} \in ClassicQuorum : \forall p_i \in \mathcal{CQ} : \exists\langle \bar{c}, \bar{\mathcal{T}} < \mathcal{T}, -, accepted, -, -\rangle \in H_i \Rightarrow$

$\neg$SlowDecisionFromRetryFP$[\bar{c}, \bar{\mathcal{T}}, \bar{\mathcal{P}red}, \bar{\mathcal{B}}]$ in retry phase started in fast propose phase.

This is a contradiction.

$\square$

Lemma 7. $\forall c, \bar{c}$, (SlowDecisionFromProposal$[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge$ SlowDecisionFromRetrySP$[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge c \sim \bar{c} \Rightarrow \bar{c} \in \mathcal{P}red$).

Proof. Assume that,

SlowDecisionFromProposal$[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge$

SlowDecisionFromRetrySP$[\bar{c}, \bar{\mathcal{T}}, \bar{\mathcal{P}red}, \bar{\mathcal{B}}] \wedge \bar{\mathcal{T}} \wedge \mathcal{T} \wedge \bar{c} \notin \mathcal{P}red \wedge \bar{c} \sim c$

$\bar{c} \notin \mathcal{P}red \Rightarrow$

$\exists \mathcal{B}' \leq \mathcal{B}, \exists \mathcal{CQ} \in ClassicQuorums : \forall p_j \in \mathcal{CQ}, \exists\langle c, \mathcal{T}, \mathcal{P}red, fast\text{-}pending, \mathcal{B}', -\rangle \in H_j \wedge \bar{c} \notin \mathcal{P}red_j \wedge (\exists\langle \bar{c}, \mathcal{T}_2 < \mathcal{T}, -, -, -, -\rangle \in H_j \Rightarrow$

$(\exists\langle \bar{c}, \mathcal{T}_2 < \mathcal{T}, -, fast\text{-}pending, -, -\rangle \in H_j \vee$Wait$(\bar{c}, \mathcal{T}_2 < \mathcal{T}, p_j) \vee \exists\langle \bar{c}, \mathcal{T}_2 < \mathcal{T}, -, rejected, -, -\rangle \in H_j)$

$\wedge(\nexists\langle \bar{c}, \mathcal{T}_2 < \mathcal{T}, -, accepted, -, -\rangle \in H_j \wedge \nexists\langle \bar{c}, \mathcal{T}_2 < \mathcal{T}, -, slow\text{-}pending, -, -\rangle \in H_j)$

This is a contradiction because

SlowDecisionFromProposal$(\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}}) \Rightarrow$

$\forall \mathcal{CQ} \in ClassicQuorums : \exists p_k \in \mathcal{CQ} : \exists\langle \bar{c}, \bar{\mathcal{T}}, \mathcal{P}red_k, accepted, \bar{\mathcal{B}}, -\rangle \in H_k \wedge \neg$Wait$(\bar{c}, \bar{\mathcal{T}}, p_k) \wedge \nexists\langle \bar{c}, \bar{\mathcal{T}}, -, rejected, \bar{\mathcal{B}}, -\rangle \in H_k$

$\square$

Lemma 8. $\forall c, \bar{c}$, (SlowDecisionFromRetry$[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge$ (SlowDecisionFromRetry$[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \overline{\mathcal{B}}] \vee$ SlowDecisionFromProposal$[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \overline{\mathcal{B}}]) \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge c \sim \bar{c} \Rightarrow \bar{c} \in \mathcal{P}red$).

Proof. Assume that,

SlowDecisionFromRetry$(c, \mathcal{T}, \mathcal{P}red, \mathcal{B}) \wedge$ SlowDecisionFromRetry$[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \wedge \bar{c} \notin \mathcal{P}red \wedge \bar{c} \sim c$

SlowDecisionFromRetry$(c, \mathcal{T}, \mathcal{P}red, \mathcal{B}) \Rightarrow$

$\exists \mathcal{CQ} \in ClassicQuorums, \forall p_i \in \mathcal{CQ} : \exists\langle c, \mathcal{T}, \mathcal{P}red_i, accepted, \mathcal{B}, \bot\rangle \in H_i \wedge \bar{c} \notin \mathcal{P}red_i$

$\wedge(\nexists\langle \bar{c}, \bar{\mathcal{T}}, -, -, -, -, \rangle \in H_i \vee \exists\langle \bar{c}, \bar{\mathcal{T}}, -, rejected, -, -, \rangle \in H_i) \Rightarrow$

$\forall \mathcal{CQ} \in ClassicQuorums, \exists p_i \in \mathcal{CQ} : \nexists\langle \bar{c}, \bar{\mathcal{T}}, -, -, -, -\rangle \in H_i \vee \exists\langle \bar{c}, \bar{\mathcal{T}}, -, rejected, -, -\rangle \in H_i$

$\Rightarrow \neg$SlowDecisionFromRetry$[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}}]$

This a contradiction.

□

Theorem 1. $\forall c, \bar{c}, ($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. The proof follows from the Lemmas 1–8

□

Lemma 9. $\forall c\ (\exists \mathcal{B},$ SlowDecisionFromProposal$[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge$
$\forall \bar{c} \in \mathcal{P}red,$ 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. SlowDecisionFromProposal$[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge \exists \bar{c} \in \mathcal{P}red :$ Decided$[\bar{c}, \bar{\mathcal{T}}, -, -] \wedge \bar{\mathcal{T}} < \mathcal{T} \Rightarrow \forall \mathcal{B}' \geq \mathcal{B},$ Decided$[c, \mathcal{T}, \mathcal{P}red', \mathcal{B}'] \wedge \bar{c} \in \mathcal{P}red'$

SlowDecisionFromProposal$[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \Rightarrow$

$\forall \mathcal{CQ} \in ClassicQuorums, \exists p_i \in \mathcal{CQ} : \exists\langle c, \mathcal{T}, \mathcal{P}red, slow\text{-}pending/stable, \mathcal{B}, -\rangle \in H_i \vee \langle c, \mathcal{T}, \mathcal{P}red, slow\text{-}pending/stable, \mathcal{B}', -\rangle \in H_i$

□

Lemma 10. $\forall c\ (\exists \mathcal{B},$ SlowDecisionFromRetry$[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge$
$\forall \bar{c} \in \mathcal{P}red,$ 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. SlowDecisionFromRetry$[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \Rightarrow$

$\forall \mathcal{B}' \geq \mathcal{B}, \forall \mathcal{CQ} \in ClassicQuorums, (\exists p_i \in \mathcal{CQ} : \exists\langle c, \mathcal{T}, -, accepted, \mathcal{B}', -\rangle \in H_i) \wedge (\exists p_k \in \mathcal{CQ} : \exists\langle \bar{c}, \bar{\mathcal{T}}, -, -, -, -\rangle \in p_k)$

□

Lemma 11. $\forall c\ (\exists \mathcal{B},$ FastDecision$[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge \forall \bar{c} \in \mathcal{P}red,$ 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. FastDecision$[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge \exists \bar{c} \in \mathcal{P}red :$ Decided$[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}}] \wedge \bar{\mathcal{T}} < \mathcal{T} \Rightarrow \forall \mathcal{B}' \geq \mathcal{B},$ Decided$[c, \mathcal{T}, \mathcal{P}red', \mathcal{B}'] \wedge \bar{c} \in \mathcal{P}red'$

$\textsc{Decided}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}}] \wedge$

$\forall \mathcal{CQ} \in ClassicQuorums, \nexists p_h \in \mathcal{CQ} : \exists \langle \bar{c}, \bar{\mathcal{T}}, -, accepted/slow\text{-}pending/stable, -, - \rangle \in H_h$

$\Rightarrow \exists \mathcal{FQ} \in FastQuorums : \forall p_i \in \mathcal{FQ}, \exists \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red_i}, fast\text{-}pending, \bar{\mathcal{B}}', - \rangle \in H_i \wedge$
$\nexists \langle \bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red_i}, rejected, \bar{\mathcal{B}}, - \rangle \in H_i \wedge \neg \textsc{Wait}(\bar{c}, \bar{\mathcal{T}}, p_i)$

$\Rightarrow \exists \mathcal{FQ} \in FastQuorums : \forall p_i \in \mathcal{FQ}, \exists \langle c, \mathcal{T}, \mathcal{P}red, stable, \mathcal{B}', - \rangle \in H_k \vee$
$(\exists \langle c, \mathcal{T}, \mathcal{P}red_i, fast\text{-}pending, \mathcal{B}', - \rangle \in H_i \wedge \bar{c} \in \mathcal{P}red_i) \vee \nexists \langle c, \mathcal{T}, \mathcal{P}red_i, -, -, - \rangle \in H_i$

$\textsc{FastDecision}[c, \mathcal{T}, \mathcal{P}red, \mathcal{B}] \wedge \textsc{Decided}[\bar{c}, \bar{\mathcal{T}}, \overline{\mathcal{P}red}, \bar{\mathcal{B}}] \wedge$
$\forall \mathcal{CQ} \in ClassicQuorums, \nexists p_k \in \mathcal{CQ} : \exists \langle \bar{c}, \bar{\mathcal{T}}, -, accepted/slow\text{-}pending/stable, -, - \rangle \in H_h$

$\Rightarrow \exists \mathcal{FQ}_1, \mathcal{FQ}_2 \in FastQuorums : \forall p_i \in \mathcal{FQ}_1 \cap \mathcal{FQ}_2, (\exists \langle c, \mathcal{T}, \mathcal{P}red_i, fast\text{-}pending, \mathcal{B}, \bot \rangle \in H_i \wedge \bar{c} \in \mathcal{P}red_i) \vee (\exists \langle c, \mathcal{T}, \mathcal{P}red_i, stable, \mathcal{B}', - \rangle \in H_i)$

Since $\forall \mathcal{FQ}_1, \mathcal{FQ}_2 \in FastQuorums, \forall \mathcal{CQ} \in ClassicQuorums, \mathcal{FQ}_1 \cap \mathcal{FQ}_2 \cap \mathcal{CQ} \neq \emptyset$

$\Rightarrow \forall \mathcal{CQ} \in ClassicQuorums, \exists \mathcal{FQ} \in FastQuorums :$

$\exists p_i \in \mathcal{CQ} \cap \mathcal{FQ} : (\exists \langle c, \mathcal{T}, \mathcal{P}red_i, fast\text{-}pending, \mathcal{B}, \bot \rangle \in H_i \wedge \bar{c} \in \mathcal{P}red_i)$

$\vee (\exists \langle c, \mathcal{T}, \mathcal{P}red_i, stable, \mathcal{B}', - \rangle \in H_i) \forall \mathcal{B}' \geq \mathcal{B}, \exists pk : \exists \langle c, \mathcal{T}, \mathcal{P}red_k, fast\text{-}pending, \mathcal{B}', \top \rangle \in H_k \Rightarrow \bar{c} \in \mathcal{P}red_k$

Therefore

$\forall \mathcal{B}' \geq \mathcal{B}, \forall \mathcal{CQ} \in ClassicQuorums : \exists p_i : \exists \langle c, \mathcal{T}, -, fast\text{-}pending/slow\text{-}pending/stable, \mathcal{B}', - \rangle \wedge$
$\nexists \langle c, \mathcal{T}, -, rejected, \mathcal{B}', - \rangle \in H_i \wedge$

(

$\exists p_j \in \mathcal{CQ} : \exists \bar{c}, \bar{\mathcal{T}}, -, accepted/slow\text{-}pending/stable, -, - \rangle \in H_k \vee$

$\exists p_k \in \mathcal{CQ} : \exists c, \mathcal{T}, \mathcal{P}red_k, fast\text{-}pending, \mathcal{B}', \top \rangle \in H_k \wedge \bar{c} \in \mathcal{P}red_k \vee$

$\nexists \mathcal{FQ} \in FastQuorums, \nexists \mathcal{CQ}' \in ClassicQuorums : \forall p_k \in \mathcal{FQ} \cap \mathcal{CQ}',$
$\exists \langle c, \mathcal{T}, \mathcal{P}red_k, fast\text{-}pending, \mathcal{B}, \bot \rangle \in H_k \wedge$

$\bar{c} \in \mathcal{P}red_k \wedge \nexists \langle c, \mathcal{T}, \mathcal{P}red, stable, \mathcal{B}, \bot \rangle \in H_k$

)

□

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

Proof. The proof follows from the Lemmas 9–11 □

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Replication is a widely adopted method to scale applications to millions of users, while ensuring availability. Among the various replication techniques, State Machine Replication has been extensively studied in literature due to its effectiveness in masking failures as every issued is executed by each replica of the distributed system. The Consensus problem abstracts away the coordination required to reach an agreement among replicas regarding the final execution order of the commands to ensure that every replica in the system reach the same final state. Chapter 2 overviewed Paxos [10], the most popular consensus algorithm; Multi-Paxos, the most deployed consensus algorithm; and Mencius [14], a multi-leader based consensus algorithm. All these protocols find a total order among all the commands submitted to the system, therefore limiting concurrency. Instead, Generalized Consensus [19] proposes ordering only conflicting commands and exploit the commutativity property of non-conflicting commands in order to improve performance. EPaxos [11] was the first algorithm to implement Generalized Consensus for solving consensus in the wide area. Alvin [12] and M^2Paxos [13] are also examples of Generalized Consensus implementations.

This thesis presents a low-latency consensus algorithm for geographically replicated systems. The main motivation of this work is the performance degradation in existing implementations of Generalized Consensus when the amount of conflicting commands increases. For instance, an EPaxos replica can decide a command c in the fast path only when no other replica proposes a command conflicting with c; otherwise, it should also take the slow path to decide c, resulting in a total of four communication delays. On the other hand, CAESAR uses a combination of logical timestamps and wait condition to increase the probability of delivering a command in the fast path even under the presence of conflicts. The extensive evaluation presented in Chapter 5 confirms CAESAR's effectiveness in geo-replication. This makes CAESAR very suitable for building tomorrow's low-latency geo-replicated ser-

vices while providing strong consistency. Providing strong consistency simplifies application complexity as developers can now treat the system of replicas as a single instance and not worry about the underlying details.

7.2 Lessons Learnt

Consensus can be broadly classified as total order based and Generalized Consensus. Total order based consensus protocols such as Paxos, Multi-Paxos and Mencius are easier and simpler to implement compared to Generalized Consensus protocols such as EPaxos, Alvin, and M^2Paxos . However, the inherent disadvantage of total order based protocols is that they are less prone to scale since they do not exploit parallelism exhibited by commuting commands, unlike Generalized Consensus. On the other hand, Generalized Consensus protocols are complex and harder but tend to scale very well. Moreover, Generalized consensus implementations pre-Caesar provide their best performance when the workload consists of mostly non-conflicting commands (i.e., $< 5\%$ conflicting commands). Caesar breaks this limitation and boosts the performance even under workloads with substantial amount of conflicting commands (i.e., $< 40\%$). However, when the workload's conflict rate goes beyond 40%, it is best to use a total order based protocol since the performance provided by its simplistic ordering scheme surpasses the performance provided by Generalized Consensus protocols at such high conflict rates. This is because Generalized Consensus protocols have a substantial overhead and the lack of parallelism that can be exploited from the workload exposes this overhead thus reducing the overall performance of the system.

The idea of exploiting commutativity for improving the performance have been explored in other areas as well, notably Software Transaction Memory [29] and Operating Systems [30]. The authors in [29] propose a technique called Transaction Boosting that uses the commutativity property of objects to determine conflicts and synchronize operations on a per object basis. The commutativity property has also been exploited in implementing scalable system call APIs. The authors of [30] propose a tool named Commuter that takes as input high-level interface models and generates tests of operations that commute. This enables implementing the interfaces in a way that could naturally scale. The idea of exploiting commutativity in Caesar also helps it scale, and thus provide higher performance. Moreover, the idea of taking fast and slow paths have been explored to implement wait-free data structures [31]. The authors in [31] implement a wait-free linked list that runs lock-free fast paths under no contention and switches to wait-free slow path upon contention. This is similar to how Caesar works, except that Caesar's novelties allows it take fast path even under conflicts.

7.3 Future Work

The contribution of this thesis has immense scope for future work:

The evaluation in Chapter 5 presents the performance of the Caesar and its competitors for a handful of conflict rates. This is adequate to understand the trend in performance as conflict rate increases. However, it is vital to understand the protocol's behavior when injected with a real world workload. The common case workload may be between any two pairs of the presented conflicted rates, and this requires testing the conflict rates within the conflict window. The task of finding a real workload trace, analyzing it to find the conflict rate, and finding the performance of the system at that conflict rate is left for future work.

Moreover, the evaluation was performed on a five-node setup deployed in five AWS regions using virtual machines with 8 vCPUs and 32GB of memory. The performance of the system depends on the deployment setup as well as the data center and network infrastructures. There are many choices of cloud providers today and each of them have unique features. For instance, Google Cloud Platform [32] has dedicated interconnects between their data centers (globally), thus they provide some service guarantees on the network latency. Amazon Web Services, on the other hand, use the internet to establish inter-region connections. Thus, it is vital to understand the impact of infrastructure (e.g. machine configuration) as well as the replica placement on the performance of the system. This is left for future work.

Another natural extension to the work mentioned in the previous paragraph is exploring data center locality. Cloud providers colocate multiple data centers within a geographic region to ensure availability and low latency during datacenter failures. The idea is to place instances of the application on multiple colocated data centers rather than a single one, such that when one of the data center fails, another data center in the region can serve users without performance loss. It is crucial to observe Caesar's behavior when replicas are placed in a combination of data centers that are colocated in addition to those in different regions.

Another possible future work is to minimize the long tail latencies. Caesar can easily exhibit a latency distribution with a large standard deviation since decisions can be made after one or two round trips of communication in addition to the wait condition, if necessary. For applications with a certain Service Level Agreement (SLAs), care has to be taken to ensure that requests are mostly executed within a certain time. This is an interesting direction, especially with the prominence of the cloud computing. Cloud providers charge their customers for a certain SLA, and therefore when the cloud providers fail to meet their target SLA, they pay a penalty back to the customer [33]. Thus, it is very important that large tail latencies are minimized.

Bibliography

[1] F. Lardinois, “Gmail now has more than 1b monthly active users,” February 2016. [Online]. Available: https://techcrunch.com/2016/02/01/gmail-now-has-more-than-1b-monthly-active-users/

[2] “Facebook company info,” 2016. [Online]. Available: http://newsroom.fb.com/company-info/

[3] Amazon Inc., “Amazon Web Services,” 2016, url: http://aws.amazon.com/.

[4] (2010, June) amazon.com’s journey to the cloud. [Online]. Available: https://s3.amazonaws.com/aws001/trailhead/MigratingAmazonComToAWS.PDF

[5] “Completing the netflix cloud migration,” February 2016. [Online]. Available: https://media.netflix.com/en/company-blog/completing-the-netflix-cloud-migration

[6] P. Bailis and A. Ghodsi, “Eventual consistency today: Limitations, extensions, and beyond,” *Communications of the ACM*, vol. 56, no. 5, pp. 55–63, 2013.

[7] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, “Don’t settle for eventual: scalable causal consistency for wide-area storage with cops,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles.* ACM, 2011, pp. 401–416.

[8] B. Charron-Bost and A. Schiper, “Uniform Consensus is Harder Than Consensus,” *J. Algorithms*, vol. 51, no. 1, pp. 15–37, Apr. 2004.

[9] L. Lamport, “The Part-time Parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[10] ——, “Paxos made simple,” *ACM Sigact News*, 2001.

[11] I. Moraru, D. G. Andersen, and M. Kaminsky, “There is More Consensus in Egalitarian Parliaments,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP ’13. ACM, 2013, pp. 358–372.

[12] A. Turcu, S. Peluso, R. Palmieri, and B. Ravindran, "Be General and Don't Give Up Consistency in Geo-Replicated Transactional Systems," in *Proceedings of the 18th International Conference on Principles of Distributed Systems*, ser. OPODIS '14, 2014, pp. 33–48.

[13] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran, "Making fast consensus generally faster," in *DSN*, 2016.

[14] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building Efficient Replicated State Machines for WANs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USENIX Association, 2008, pp. 369–384.

[15] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Database replication techniques: a three parameter classification," in *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*, 2000, pp. 206–215.

[16] (2016, September) Overview: Sql database active geo-replication. [Online]. Available: https://docs.microsoft.com/en-us/azure/sql-database/sql-database-geo-replication-overview

[17] Db instance replication. Accessed: 2017-01-25. [Online]. Available: http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Overview.Replication.html

[18] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/98163.98167

[19] L. Lamport, "Generalized Consensus and Paxos," Microsoft Research, Tech. Rep. MSR-TR-2005-33, March 2005.

[20] 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.

[21] W. Wei, H. T. Gao, F. Xu, and Q. Li, "Fast mencius: Mencius with low commit latency," in *Proceedings of the IEEE INFOCOM 2013, Turin, Italy, April 14-19, 2013*, 2013, pp. 881–889.

[22] 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.

[23] 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.

[24] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming.* Springer, 2006.

[25] 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.

[26] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper, "Jpaxos: State machine replication based on the paxos protocol," 2011. [Online]. Available: https://github.com/JPaxos/JPaxos

[27] Epaxos. [Online]. Available: https://github.com/efficient/epaxos

[28] hyflow-go: a transaction execution framework in go. [Online]. Available: https://bitbucket.org/talex/hyflow-go

[29] M. Herlihy and E. Koskinen, "Transactional boosting: A methodology for highly-concurrent transactional objects," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 207–216. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345237

[30] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, pp. 10:1–10:47, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2699681

[31]

[32] Google cloud platform. [Online]. Available: https://cloud.google.com/

[33] Amazon ec2 service level agreement. [Online]. Available: https://aws.amazon.com/ec2/sla/