FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Towards Causal Consistency in Read-Heavy Cloud-Native Systems

Diana Cristina Amaral de Freitas

Mestrado em Engenharia Informática e Computação

Supervisor: Tiago Boldt Pereira de Sousa

External Supervisor: Paul deGrandis

July 20, 2023

Towards Causal Consistency in Read-Heavy Cloud-Native Systems

Diana Cristina Amaral de Freitas

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Carlos Miguel Ferraz Baquero-Moreno

Referee: Prof. Tiago Boldt Pereira de Sousa

Referee: Prof. João Coelho Garcia

July 20, 2023

Abstract

In geo-replicated distributed systems, data is redundantly stored across nodes at different geographical sites, increasing fault tolerance and reducing latency for end users.

With updates being concurrently issued across sites, replicas should converge to a consistent view of the data, which leads toward adopting fine-tuned consistency models, namely causal consistency (CC). CC captures the potential causality between events, guaranteeing that all replicas agree on the order of causally related operations. When coupled with read-only transactions (ROTs), CC can improve over the limitations of eventual consistency (EC), mainly its lack of ordering guarantees, even when the data is partitioned across servers.

ROTs are necessary to extract a consistent view across partitions. However, they incur additional coordination overhead compared to non-transactional reads, making it crucial to optimize their performance, especially given the prevalence of read operations in several real-world applications.

In this regard, the literature has focused on improving the latency, throughput, and data visibility of ROTs within causally consistent key-value data stores. In particular, it has identified the properties of ROT algorithms that maximize their performance.

Despite the relevancy of these novel results in the scope of read-heavy systems, existing solutions for assembling geo-replicated causally consistent systems (1) focus narrowly on the algorithmic design to ensure causality within key-value data stores, lacking a generic architecture for applying CC to read-heavy systems, (2) do not fully leverage the benefits of existing cloud storage infrastructure, and (3) overlook the auditing properties of their designs, limiting developers' ability to reason about the system's behavior.

With this in mind, there is a need to bridge the gap between academic research and industry requirements by transposing the literature's findings into a reference architecture for delivering CC in read-heavy systems and determining how it can be realized to enable auditing.

To that end, the present study aims to demonstrate that:

> *There exists a reference architecture that (1) manifests the ideal properties of geo-replicated causally consistent read-heavy systems, (2) upgrades the consistency guarantees of existing cloud storage services, and (3) enables value semantics, thereby facilitating auditing and enabling developers to reason about the system's state and data at a point in time.*

To validate the hypothesis, we first survey the state of the art of causally consistent distributed systems, identifying the properties that maximize ROTs' performance and the strategies that enable those properties. Then, we propose a reference architecture for read-heavy systems that enables CC atop existing cloud storage services and manifests the properties identified in the literature. Finally, we realize and validate the reference architecture above Amazon Simple Storage Service (Amazon S3), demonstrating how enabling *value semantics* in our implementation increases auditability.

This work's empirical validation suggests that data staleness is the main trade-off of upgrading existing storage services with CC. This trade-off can be managed by adjusting the periodicity at which new versions get persisted and retrieved from the storage layer. Moreover, the results confirm the system's scalability upon increased read load, affirming the suitability of the reference architecture for read-heavy systems. Finally, our experimental verification demonstrates how enabling *value semantics* improves the system's auditability, allowing developers to observe its state at a point in time.

Keywords: Read-heavy, Causal Consistency, Performance, Distributed Systems, Geo-replication, Read-only transactions, Value Semantics

ACM Classification: Computer systems organization → Distributed Architectures, Software and its engineering → Distributed systems organizing principles

Resumo

Num sistema distribuído geo-replicado, os dados são armazenados de forma redundante em nós localizados em diferentes regiões geográficas, permitindo uma maior tolerância a falhas e reduzindo a latência de acesso aos dados.

Com operações de escrita a ocorrerem em simultâneo em diferentes regiões, as réplicas devem convergir para uma visão consistente dos dados, levando à adoção de modelos de consistência tais como a consistência causal. A consistência causal capta a potencial causalidade entre eventos, garantindo que todas as réplicas concordam na ordem das operações causalmente relacionadas. Quando associada a transações de leitura, a consistência causal permite ultrapassar as limitações da consistência eventual, principalmente a falta de garantias no que diz respeito à ordenação dos eventos, mesmo quando os dados estão repartidos pelos servidores.

As transações de leitura são necessárias para obter uma visão consistente dos dados armazenados nas diferentes partições. No entanto, elas impõe uma sobrecarga de coordenação adicional em comparação com as leituras não transacionais, sendo crucial otimizar o seu desempenho, especialmente dada a prevalência de operações de leitura em várias aplicações.

Neste sentido, a literatura tem procurado melhorar a latência, taxa de transferência e visibilidade dos dados das transações de leitura em bases de dados de chave-valor causalmente consistentes, tendo, em particular, identificado as propriedades dos algoritmos de transações de leitura que permitem maximizar o seu desempenho.

Apesar da relevância destes resultados no contexto de sistemas de leitura intensiva, as soluções existentes para a implementação de sistemas geo-replicados causalmente consistentes (1) focam-se maioritariamente nos algoritmos utilizados para garantir causalidade em bases de dados de chave-valor, não apresentando uma arquitetura genérica para construir sistemas de leitura intensiva causalmente consistentes, (2) não tiram partido dos atuais serviços de armazenamento na nuvem e (3) dificultam a análise do comportamento do sistema ao não potenciarem as propriedades de auditoria inerentes à sua arquitetura.

Tendo isto em conta, é necessário colmatar a lacuna entre a investigação académica e os requisitos da indústria, transpondo os resultados da literatura para uma arquitetura de referência que garanta consistência causal em sistemas de leitura intensiva e determinando de que forma esta pode ser implementada de modo a tornar o sistema auditável.

Para tal, esta dissertação visa demonstrar que:

> *Existe uma arquitetura de referência que (1) manifesta as propriedades ideais de sistemas geo-replicados causalmente consistentes de leitura intensiva, (2) amplifica as garantias de consistência dos atuais serviços de armazenamento na nuvem e (3) permite a utilização de semântica de valor, tornando o sistema auditável e possibilitando a análise do seu estado e dados num ponto no tempo.*

Para validar a hipótese, começamos por analisar o estado da arte no âmbito de sistemas distribuídos causalmente consistentes, identificando as propriedades que maximizam o desempenho das transações de leitura e as estratégias que permitem garantir essas propriedades. Em seguida,

propomos uma arquitetura de referência para sistemas de leitura intensiva que garante consistência causal sobre os serviços de armazenamento na nuvem existentes e manifesta as propriedades identificadas na literatura. Por fim, aplicamos e validamos a arquitetura de referência sobre o Amazon S3, demonstrando de que forma a utilização de *semântica de valor* na nossa implementação aumenta a sua auditabilidade.

A validação empírica deste trabalho sugere que o fator mais afetado ao amplificar os serviços de armazenamento existentes com consistência causal é a visibilidade dos dados. Esta desvantagem pode ser gerida ajustando a periodicidade com que as novas versões são armazenadas e recuperadas da camada de armazenamento. Os resultados confirmam ainda a escalabilidade do sistema aquando do aumento da carga de leitura, afirmando a adequação da arquitetura de referência para sistemas de leitura intensiva. Finalmente, a nossa verificação experimental demonstra que a utilização de *semântica de valor* melhora a auditabilidade do sistema, permitindo aos programadores observar o seu estado num determinado momento.

Acknowledgements

First, I would like to thank my supervisor, Professor Tiago Boldt de Sousa, for his guidance and advice, which were essential for the success of this work. Thank you for providing me with the opportunity to work closely with Kevel and for consistently pushing me to do better.

I am deeply thankful to my external supervisor, Paul deGrandis, for providing me with the knowledge that allowed me to accomplish this work, for answering my endless questions, and for the encouragement and confidence in my abilities. His knowledge, advice, and guidance have been crucial for the success of this work.

I would also like to thank João Azevedo for his support during the initial stages of my research, which significantly influenced the direction of this work.

I am genuinely grateful to my family for their support through all the ups and downs of my academic journey. Thank you for believing in my abilities, supporting my decisions, and consistently insisting that I take a break and get fresh air when needed.

I also want to thank my friends, especially Mariana Ramos, Rita Silva, and Raquel Sepúlveda, who have been there for me from the beginning of this journey. Their constant support and friendship have been irreplaceable. A special thank you goes to Juliane Marubayashi and Diogo Samuel Fernandes, whose encouragement, cheerfulness, and exceptional companionship made this journey more meaningful.

I also want to express my gratitude to João Lage for his patience, support, and faith in me, not only throughout this process but also through the last couple of years. Thank you for always listening to my extensive monologues and for being there for me every step of the way.

Lastly, I cannot forget to thank Muni, my dog, for her irreplaceable companionship and for forcing me to step outside my house every day, even if only for 15 minutes.

Diana Freitas

"If it scares you,
it might be a good thing to try."

Seth Godin

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

2PC Two-Phase Commit. 35, 40, 41, 43–45, 68, 69, 73, 74

Amazon EC2 Amazon Elastic Compute Cloud. 99, 101, 112, 118, 119, 123, 127–129

Amazon S3 Amazon Simple Storage Service. i, iv, 65, 66, 99–101, 106–114, 118–122, 125, 127–130, 132, 134, 137–142

ATAM Architectural Tradeoff Analysis Method. x, 65–67, 75, 85, 134

AWS Amazon Web Services. 65, 99, 100, 109, 112, 118, 120, 128, 132, 134, 137, 139

CANToR Client-Assisted Nonblocking Transactional Reads. 117, 131, 136

CC Causal consistency. i, ii, x, 2–4, 14–18, 22, 25, 26, 28–32, 36–38, 44, 51–55, 58, 60–69, 71, 77, 86–88, 90–96, 98, 101, 102, 112, 114, 118, 121–123, 128, 131–140, 142

CC+ Causal+ consistency. 15, 17, 22, 29, 30, 33, 34, 76, 114, 115, 133, 142

CRDT Conflict-Free Replicated Data Types. 33, 34, 46, 96

DC Data center. 29–40, 42–44, 46–52, 56–59, 68–70, 73, 74, 89, 90, 96, 97, 135, 136

DHT Distributed Hash Table. 141

EC Eventual consistency. i, x, 2, 13–15, 22, 28, 60, 62, 127, 133, 142

ECDS Eventually Consistent Data Store. 54, 58, 68–71, 75, 76, 79, 135, 137

gRPC Google Remote Procedure Call. 99–101, 103, 104, 110, 112, 120, 152

GST Global Stable Time. 43, 70

HLC Hybrid Logical Clock. x, 10–12, 15, 22, 40–43, 46–48, 52, 53, 55, 68, 71, 78, 88–90, 95, 97, 99–102, 104–106, 108, 111–113, 135–137, 141

LST Local Stable Time. 40

MVCC Multi-Version Concurrency Control. 21

NTP Network Time Protocol. 7, 10, 90

OCC Optimistic Causal Consistency. 38

ROT Read-only transaction. i, x, xi, 2–4, 6, 17–20, 22, 23, 25, 26, 28–30, 32, 33, 36, 39, 41–53, 56–58, 61–63, 65, 68, 69, 71–79, 81–84, 87–91, 93, 95–97, 99–103, 110–117, 123–126, 128, 129, 131, 133–140, 142, 152, 153

RST Remote Stable Time. 40, 41

SLA Service level agreements. 3, 56

SLO Service level objectives. 56

TAI International Atomic Time. 7

TCC Transactional causal consistency. 17, 28, 34, 40, 42, 43, 46

UST Universal Stable Time. x, 43, 44, 46, 68, 70, 75, 90

UT Universal Time. 7

UTC Coordinated Universal Time. 7

WOT Write-only transaction. x, 17, 30, 44, 45, 48, 73, 74, 135, 140

YCSB Yahoo! Cloud Serving Benchmark. 34

Chapter 1

Introduction

Contents

This chapter introduces the scope of this dissertation. Firstly, sections 1.1 and 1.2, respectively, describe the context and motivation of this work. Then, section 1.3 summarizes the problem under study, and section 1.4 details the main goals of this dissertation. Finally, section 1.5 describes how this document is structured.

1.1 Context

A distributed system consists of a set of nodes that coordinate through the exchange of messages, appearing to their users as a single cohesive unit [47, 94]. By replicating data across nodes at possibly different geographical sites (geo-replication), they can provide fault tolerance and bring data closer to the user, thus reducing latency [55].

With replication, data writes can simultaneously occur across nodes, making it necessary to keep replicas consistent — "when one copy is updated we need to ensure that the other copies are updated as well; otherwise the replicas will no longer be the same" [94, p.357]. For that purpose, consistency models apply weaker or stricter restrictions on the values returned by read operations, offering various guarantees.

Stronger consistency models impose stricter ordering constraints but are incompatible with low latency [2] and, in the presence of network partitions, with high availability [32]. In this regard, especially with the rise of cloud computing, weaker consistency models have been an attractive choice for implementing distributed systems [95].

One such model is eventual consistency (EC). In the absence of updates, EC guarantees that all reads will eventually return the last value [95]. However, it does not guarantee safety, and thus, until replicas converge, reads may not reflect the operations in the expected order or the last writes.

Causal consistency (CC) is another weak consistency model that ensures replicas process potentially causally related events in the same order [94]. The appeal of CC stems from its ability to address the limitations of EC, particularly its lack of ordering guarantees, and to minimize the coordination overhead of stronger consistency models. Moreover, in the presence of network partitions, CC is the strongest consistency model compatible with high availability [62]. Lastly, by capturing the causality relationship between events, CC facilitates reasoning about distributed computations [78], ensuring users observe events in the order they intuitively expect [55].

Regardless of its benefits, CC does not overcome all the challenges of real-world workloads where data does not fit on a single machine [55]. For example, Meta's geographically distributed data store serves over ten billion requests per second on a changing data set of many petabytes, making it necessary to shard the data across nodes to scale capacity and throughput [15, 18]. With data spread across servers, the client's reads may reach different servers, possibly resulting in an inconsistent view of the data, which led previous studies to rely on read-only transactions (ROTs) to unify the view of data across shards. Even though there are many techniques to implement ROTs, to assure availability, a common strategy is to maintain multiple versions of the data and perform all read operations of the ROT from the same logical time [55].

ROTs, however, introduce an extra coordination overhead, which may hamper the system's overall performance, especially in the large class of read-heavy systems [15, 18, 69, 92, 100].

Motivated by the importance of reducing the overhead of ROT algorithms, especially in read-dominated applications, Lu *et al.* [56] identified the properties of ROTs that make them latency-optimal. In a later work [58], the authors identified the properties that make ROTs performance-optimal (*NOC properties*) and proposed a causally consistent key-value store that attains those properties, Eiger-PORT [58], presenting further progress toward optimizing read performance in causally consistent systems.

1.2 Motivation

The prevalence of reads is evident in several real-world applications [15, 18, 69, 92, 100]. In social networks, for example, reads account for 99.7% of TAO's requests, Meta's geo-replicated data store [18]. Similarly, in Ambry [69], Linkedin's production environment, read operations make up 95% of the requests.

In online advertising, the dominance of read operations is also evident in Kevel's cloud-native ad-decision system, where decision engines perform read-only workloads to select the best subset of ads to return.

The read-heavy nature of these applications, together with the scale of data, makes it crucial to optimize ROTs, which incur additional coordination overhead than non-transactional reads but are necessary for retrieving a consistent view across shards. Optimizing ROTs is especially relevant

to deliver quality service to the end users and satisfying service requirements. In Kevel's system, in particular, the stringent service level agreements (SLAs) impose that 99% of the requests are answered within 50 milliseconds and that the availability is at least 99.999%.

Several other real-world geo-replicated read-heavy applications face similar requirements for performance and availability, which leads them to trade off strong consistency [32]. However, when data is globally replicated, partitioned across servers, and constantly being updated and accessed, ensuring low latency, high throughput, and availability while keeping replicas consistent requires trade-offs in the design of these systems.

Motivated by the importance of ROTs' performance for the large class of read-heavy systems, academic research has proposed novel algorithms capable of providing latency and performance-optimal ROTs within causally consistent key-value data stores [56, 58].

Even though this vein of academic research may unveil insights for read-heavy applications, the literature is not readily applicable to real-world systems, where data management is used to enforce each service's unique business rules and requirements and where the system's implementation may have to adhere to specific constraints. Furthermore, most designs lack auditability and, thus, make it difficult to detect faults and understand the system's behavior [53, 54, 5, 24, 8, 25, 98, 56, 4, 65, 86, 84, 23, 85, 91, 58].

With this in mind, there is a need to bridge the gap between academic research and industry requirements by transposing the results achieved in Eiger-PORT [58] and other key-values stores into a reference architecture for delivering CC in read-heavy systems and determining how it can be realized to enable auditing.

1.3 Problem

Despite the potential benefits of translating the results achieved in Eiger-PORT [58] and other related works to the broad class of read-heavy systems, existing research focuses on the algorithmic design to ensure causality, primarily within key-value data stores, lacking a system architecture that applies to a broader class of read-heavy systems.

Additionally, most systems do not harness the benefits of existing cloud storage services, namely their availability and data management capacities (e.g., accessibility, durability, and reliability), pointing toward the need to incorporate recent literature's findings in a storage-agnostic way, akin to Bailis *et al.* Bolt-on approach.

Moreover, even though versioning is inherent to several causally consistent systems, making it possible to read stale versions of the data, its auditing capabilities could be further exploited, particularly by integrating and expanding the core ideas of replicated state machines (like Viewstamped Replication [71, 51]) with stronger value semantics.

In light of these shortcomings, the present dissertation aims to assess whether the results achieved in existing causally consistent key-value stores can be extrapolated into a storage-agnostic

reference architecture for read-heavy systems that can provide CC atop existing cloud storage services. In addition, it strives to improve the auditability of these systems by leveraging *value semantics*, an interface for computation that enables the perception of values in time.

1.4 Goals

The present work seeks to research the design of geo-replicated causally consistent systems to identify the properties and design choices that enable better read performance, data visibility, and auditability and extrapolate them into a storage-agnostic reference architecture for read-heavy systems. To that end, it aims to:

Identify the ideal properties of a geo-replicated causally consistent read-heavy system:
The read-heavy nature of several real-world applications [15, 18, 69, 92, 100], together with the scale of data, point towards the relevance of identifying the properties that potentialize ROTs performance, enabling low latency and high throughput under read-dominated workloads. With this in mind, we aim to identify in the literature, namely in Eiger-PORT [58], which properties are necessary to meet these requirements. In particular, we want to understand the properties that make ROTs performance-optimal.

Determine which strategies and design choices are required to achieve those properties:
In order to idealize a reference architecture for read-heavy systems, it is crucial to review the architectures of existing systems and analyze their trade-offs. In particular, it is essential to understand how their design choices impact the performance of ROTs. In that regard, we intend to survey existing systems through the lens of the performance-optimal properties of ROTs.

Produce a storage-agnostic reference architecture for read-heavy systems that manifests those properties:
Based on our survey of causally consistent systems, we set ourselves to build a reference architecture for read-heavy systems that can ensure optimal ROT performance and minimize data staleness regardless of the specific service to which it is applied. Additionally, we aim to leverage the benefits of existing cloud storage infrastructure by incorporating recent literature findings in a storage-agnostic way.

Determine how this reference architecture can be designed to ensure *value semantics*:
Providing a reference architecture that can ensure *value semantics* is desirable because it makes it possible to implement and perceive the system as a succession of atomic states, each resulting from applying an operation to the previous one. Therefore, it enables developers to reason about the system at a point in time, simplifying programming, debugging, and decision-making.

1.5 Document Structure

This document is organized into nine chapters, structured as follows:

- Chapter 1 (p. 1), **Introduction**, outlines the context and motivation, as well as the problem under study and the main goals of this work.
- Chapter 2 (p. 6), **Background**, presents the fundamental concepts required to understand this work.
- Chapter 3 (p. 23), **State of the Art**, describes the literature review process and surveys the state of the art of causally consistent distributed systems.
- Chapter 4 (p. 60), **Problem Statement**, identifies the open problems, the scope of this work, the research questions, and the validation methodology.
- Chapter 5 (p. 67), **Preliminary Studies**, extends the literature review with an architectural analysis of existing causally consistent systems and presents the process through which the reference architecture was iteratively built and refined.
- Chapter 6 (p. 86), **Reference Architecture**, presents the reference architecture that resulted from our initial contributions and compares it with the works identified in the literature review.
- Chapter 7 (p. 99), **Reference Architecture Realization**, discusses the implementation of a prototype system that realizes the proposed architecture.
- Chapter 8 (p. 114), **Verification and Validation**, covers the verification and validation of our hypothesis.
- Chapter 9 (p. 133), **Conclusions**, summarizes the work developed throughout this dissertation, outlines its main contributions and identifies possible directions for future work.

Chapter 2

Background

Contents

This chapter outlines the fundamental concepts required to understand the present work. Firstly, section 2.1 introduces the main concepts of Distributed Systems, such as physical and logical time, replication, and consistency. Then, section 2.2 describes the general characteristics of read-heavy systems and some theorems that provide relevant results regarding the performance of ROTs. Section 2.3 explains the notion of *value semantics*. Finally, section 2.4 summarizes the key concepts mentioned in the previous sections.

2.1 Distributed Systems

In the literature, authors have proposed several definitions for distributed systems. For instance, Lamport stated that "A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages" [47, p. 558]. The notion of spatial separation between processes, however, does not imply that those processes must be geographically distant, only that the delay resulting from the exchange of messages is significant when compared with the time between events within a single process.

The present work assumes the broader definition proposed by van Steen and Tanenbaum, who described a distributed system as "a collection of autonomous computing elements that appears to its users as a single coherent system" [94, p. 2]. This definition makes no assumptions about the node topology nor imposes restrictions on how nodes are connected. Instead, it focuses on the behavior observed by the user, to whom the system must appear as a single unit, thus requiring that independent nodes coordinate in some way.

2.1.1 Time and Clocks

In a distributed system, achieving agreement on time is required to accurately timestamp events at different nodes and, thus, to know the order in which they occurred or whether they co-occurred [21, 94]. Having a common notion of time is also a requirement of many algorithms for maintaining the consistency of distributed data across nodes [21].

However, unlike in centralized systems, there is no global time reference, as each independent node has its perception of time [21, 94].

2.1.1.1 Physical Time

Most computers rely on hardware timers, often referred to as physical clocks, which use a quartz crystal oscillator to keep track of time [94]. Even though the frequency at which a crystal oscillator operates is reasonably steady, it is impacted by external factors such as temperature. As a result, it is not easy to ensure that crystals in different computers operate at the same frequency, causing clocks to drift at varying rates. Hence, different clocks exhibit different values (clock skew), so when higher precision is required, atomic clocks provide greater accuracy [94]. "Atomic clocks are based on the transitions of the cesium 133 atom, which is not only very high, but also very constant" [94, p. 304]. However, atomic clocks come at higher prices, making them unsuitable for most applications [101, 81].

In order to overcome this, clocks are synchronized through the Network Time Protocol (NTP) [1]. This protocol uses a hierarchical system of time sources, where the top level consists of high-precision timekeeping devices, such as atomic clocks, also referred to as reference clocks. It enables clock synchronization to a few milliseconds of the Coordinated Universal Time (UTC), a time standard based on the International Atomic Time (TAI) and Universal Time (UT), which accounts for the Earth's rotation. To that end, the clock skew is estimated based on the round-trip network delay. When measuring the elapsed time between two intervals, it is vital to use monotonic clocks, which apply this correction by adjusting the time rate, thus ensuring that time moves forward at a near-constant rate. If, instead, time-of-day clocks are employed, time can move backward, resulting in unexpected behavior.

Despite the possibility of synchronizing clocks, asynchrony and network unreliability make time uncertainty inevitable when using physical clocks [43]. As a result, using physical timestamps is not enough to define an ordering of events always compatible with the order observed by each node in a distributed system.

2.1.1.2 Logical Time

In light of the shortcomings of physical time, Lamport [47] introduced the *happened before* relation ($\rightarrow$), which establishes a partial order between events (i.e., a binary relation that does not specify the exact order between all pairs of events) without using physical clocks. The partial ordering specified by the *happened before* relation has its grounds on the potential causality between

events of a distributed system. In its definition, Lamport considers a system composed of a collection of processes, each consisting of a sequence of events. An event can either be a computation performed in a process or represent the sending or receiving of a message. With this in mind, the *happened before* relation is defined as follows:

(1) If events a and b occur on the same process, $a \rightarrow b$ if the occurrence of a preceded the occurrence of b;

(2) If events a and b occur in distinct processes, $a \rightarrow b$ if a is the sending of a message by a process and b is the reception of that same message in another process;

(3) Transitive Property: if a *happened before* b, and b *happened before* c, then a must have *happened before* c;

(4) If $a \not\rightarrow b$ and $b \not\rightarrow a$, then a and b are concurrent ($a \parallel b$).

"Another way to view the definition is to say that $a \rightarrow b$ means that it is possible for event a to causally affect event b. Two events are concurrent if neither can causally affect the other." [47, p. 559]

In fig. 2.1, which illustrates the *happened before* relationship between events, it is possible to observe, for example, that a_1 precedes a_2. Thus, as both events occur on node A, $a_1 \rightarrow a_2$. Moreover, assuming a_1 is the sending of a message between nodes A and B, and b_1 is the receiving of that message, we can also assert that $a_1 \rightarrow b_1$. From the transitive property, it is also possible to conclude that $a_1 \rightarrow b_2$ because $a_1 \rightarrow b_1$ and $b_1 \rightarrow b_2$. Finally, a_1 and c_1 are concurrent ($a_1 \parallel c_1$).

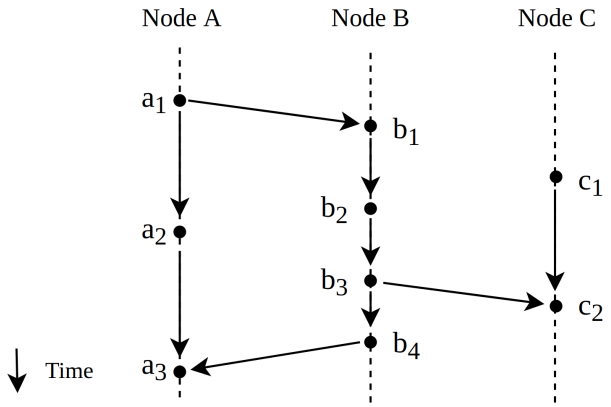


Figure 2.1: *Happened before* relationship between events. A • represents an event and an arrow from event a to b indicates that a *happened before* b (adapted from [53] and [47]).

Based on the *happened before* relation, Lamport also proposed Lamport's logical clocks, which provide a way to order events without depending on physical time. These clocks are usually implemented as numerical event counters [94]. When using Lamport's logical clocks, the following rules must be respected:

(1) Process P_i increments its logical clock before executing an event ($LC_i \leftarrow LC_i + 1$);

(2) When P_i sends a message m, it must perform the first step, incrementing its clock by one, and then set the message's timestamp t to the new value of LC_i;

(3) When P_i receives a message, it must set its clock to the maximum between its clock and the message's timestamp and then execute the first step, incrementing its clock by one ($LC_i \leftarrow \max\{LC_i, t\} + 1$).

Figure 2.2 illustrates the algorithm described above.

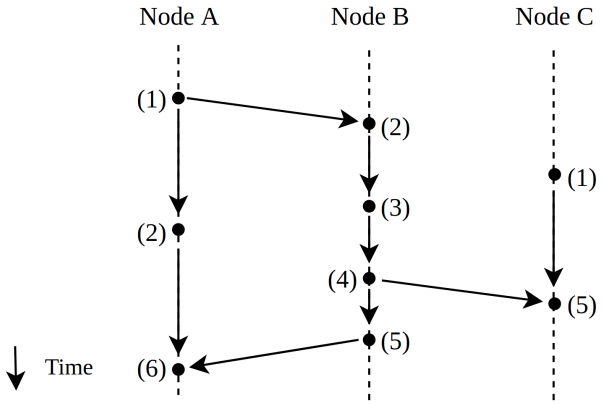


Figure 2.2: Lamport's logical clocks algorithm. A • represents an event. An arrow between two events represents the *happened before* relationship. The numbers represent the value of the logical clock of the node upon each event (adapted from [53] and [47]).

With Lamport's logical clocks, it is possible to achieve a total order of events (i.e., to establish the exact order between all pairs of events) by concatenating the process's unique identifier. Moreover, Lamport's logical clocks are coherent with the *happened before* relation, assuring that if *a happened before b* ($a \rightarrow b$), then $LC(a) < LC(b)$. However, the opposite implication does not hold, i.e., when $LC(a) < LC(b)$, it is impossible to know whether a and b are concurrent or if *a happened before b*.

To address the limitation of logical clocks, Fidge [28] and Mattern [64] concurrently introduced the concept of vector clocks. This mechanism enables partial ordering of events in a distributed system and, in contrast with logical clocks, also captures causality.

Vector clocks are data structures that allow each node to get an approximation of the global time. Each node i has a vector clock V_i (i.e., a vector with a logical clock for each node in the system), which must be initialized to zeros. On each internal event or message sent, node i must increment its entry by one ($VC_i[i] \leftarrow VC_i[i] + 1$). When a node i receives a message from node j, it increments its own entry by one ($VC_i[i] \leftarrow VC_i[i] + 1$) and updates the remaining entries by computing the maximum of its vector clock and the vector of the sender ($VC_i \leftarrow max(VC_i[k], VC_j[k]), \forall k$). Like in Lamport's timestamps, the node's vector clock is attached to each message.

Figure 2.3 illustrates the algorithm described above.

With vector clocks, it is possible to know whether two events are causally dependent or concurrent. If $VC(a) < V(b)$ (i.e., if all entries of $VC(a)$ are less or equal to the respective entries

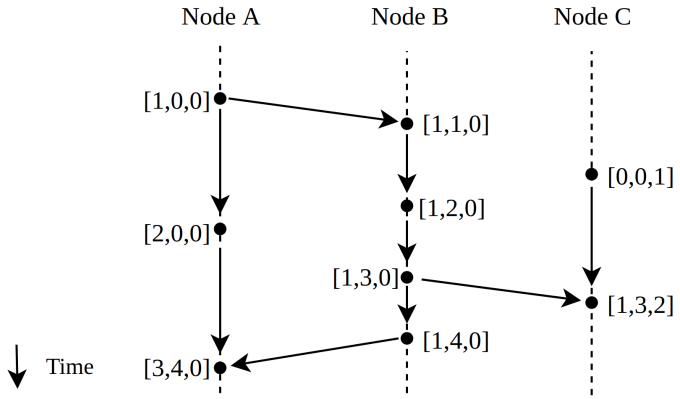


Figure 2.3: Example of the usage of vector clocks. A • represents an event. An arrow between two events represents the *happened before* relationship. The vectors near each event represent the vector clock of the node upon that event.

of $VC(b)$ and there is at least one entry for which the value in $VC(a)$ is smaller than the one in $VC(b)$) then $a \rightarrow b$. If that is not the case and the vector clocks differ (i.e., each vector contains at least one logical clock that is ahead of the other), then the events are concurrent ($a \parallel b$). In that case, a conflict resolution technique (e.g., last-writer-wins rule) can be used to order concurrent events.

Despite their advantage, vector clocks' size increases with the number of nodes in the system, introducing an extra space overhead.

In order to leverage the benefits of both hybrid and logical clocks while overcoming some of their limitations, Kulkarni, Demirbas, Madappa, *et al.* introduced HLCs [44]. "HLC captures the causality relationship like logical clocks, and enables easy identification of consistent snapshots in distributed systems" [44, p. 17]. To that end, an HLC preserves the property of logical clocks ($a \rightarrow b \implies LC(a) < LC(b)$) and therefore does not need to wait due to clock skew. "Dually, HLC can be used in lieu of physical/NTP clocks since it maintains its logical clock to be always close to the NTP clock" [44, p. 17]. The drift between HLC and the physical clock is less than the clock drift, so "it can be used to take a snapshot at a given physical time" [44, p. 31]. Finally, HLCs have the advantage of being monotonic.

The HLC algorithm shown in fig. 2.4 and exemplified in fig. 2.5 consists of the following:

(1) A node j stores two variables to keep track of logical time, l_j and c_j, which are initially set to 0. It also keeps track of physical time with pt_j;

(2) Upon a local or send event f on node j, l_j is set to the maximum of l_e and pt_j ($\max\{l_e, pt_j\}$), being e the previous event on j. This ensures that $l_j \geq pt_j$. The value of l_j is used to timestamp the event f. However, this may lead consecutive events to have the same l_j. Therefore, c_j is incremented by one and used to timestamp the event together with l_j, ensuring that $\langle l_e, c_e \rangle < \langle l_f, c_f \rangle$. If l_e differs from l_f then c_j is reset, which allows bounding the value of c_j;

(3) Upon the reception of message m on node j, l_j is set to the maximum of l_e, l_m and pt_j. If the new l_j is equal to l_m, l_e, or both, then c_j must be set. Otherwise, c_j may be reset. l_j and c_j are then used to timestamp the received event.

Initially $l.j := 0; c.j := 0$

Send or local event
$l'.j := l.j;$
$l.j := max(l'.j, pt.j);$
If $(l.j = l'.j)$ then $c.j := c.j + 1$
Else $c.j := 0;$
Timestamp with $l.j, c.j$

Receive event of message m
$l'.j := l.j;$
$l.j := max(l'.j, l.m, pt.j);$
If $(l.j = l'j = l.m)$
then $c.j := max(c.j, c.m) + 1$
Elseif $(l.j = l'.j)$ then $c.j := c.j + 1$
Elseif $(l.j = l.m)$ then $c.j := c.m + 1$
Else $c.j := 0$
Timestamp with $l.j, c.j$

Figure 2.4: HLC algorithm for node j [44].

2.1.2 Replication

In distributed systems, replication refers to the technique in which a copy of the data is kept in more than one node [21]. When these copies are stored across nodes at geographically dispersed locations, the system is usually said to be geo-replicated.

Replication can provide many of the desirable properties of distributed systems, namely:

Reliability: The ability of a system to continue operating without disruption even under network partitions (e.g., arbitrary message loss or node failures) or faults [43, 94].

Availability: The probability that a system is available and responsive at a given time, thus, ensuring that all requests eventually terminate [53, 94].

Performance: A property usually measured in terms of latency (the time that a request is waiting to be handled), response time (the time it takes to get a response to a request), and throughput (the number of operations that can be processed per unit of time) [43, 29].

Scalability: Refers to the system's ability to accommodate growth (e.g., new users, nodes, or resources) without compromising performance [43].

In sum, replication provides system scalability by enabling load balancing. Moreover, through spatial redundancy, it enables systems to continue operating regardless of the presence of network partitions or failing nodes, as nodes fail independently. It also provides higher performance

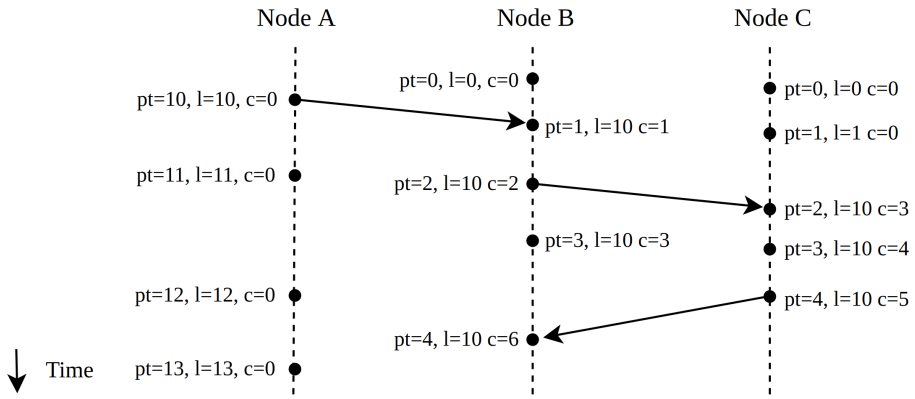


Figure 2.5: Example of the usage of HLCs. A • represents an event. An arrow between two events represents the sending of a message between nodes. Near each event are the values of the HLC in the node: l and c represent the variables used to track the logical time, whereas pt represents the value of the physical clock (based on [44]).

by placing data closer to the end users. Despite its benefits, it also raises additional challenges concerning consistency, which will be described in section 2.1.3.

2.1.2.1 Replication protocols

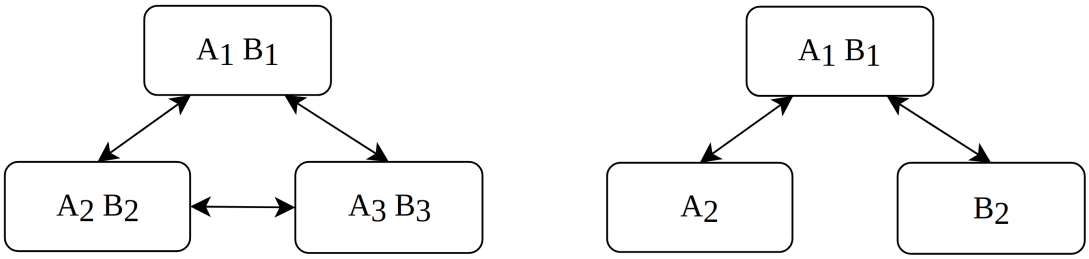
Over the years, different protocols have been proposed to leverage replication to implement fault-tolerant services.

State Machine Replication, for instance, was first introduced by Lamport [47, 46] and later explained by Schneider [79]. It consists of "a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas" [79, p. 229]. In this approach, client requests are sent to all replicas, which "independently simulate the execution of a State Machine" [47, p. 562]. Each machine, comprised of a set of state variables, applies deterministic commands that transform the state or produce some output [79]. Provided that all replicas start in the same initial state, that commands are deterministic, and that all replicas receive the same sequence of client requests, consistency is guaranteed because replicas will produce the same output [96]. Therefore, the greatest challenge is ensuring that all replicas execute the same sequence of operations. To that end, one such approach is to use atomic broadcast [22], and another is to use consensus protocols like Paxos [48, 49] and Raft [73], which aim to ensure that all nodes in the system agree on a state.

In other protocols, such as the primary-backup approach [17], the client request is only sent to a primary replica, which propagates the updates to the backups and returns the response to the client. A protocol that uses this approach is Viewstamped Replication [71, 51].

2.1.2.2 Full and partial replication

Replication can also be described concerning the degree to which data is replicated across multiple nodes in a distributed system.



(a) Full Replication with two objects, A and B.

(b) Partial Replication with two objects, A and B.

Figure 2.6: Full and Partial Replication. On the left (Full Replication), there is a copy of A and B in all three sites. There is an arrow between all replicas because an update in one replica must be propagated to all the others. On the right (Partial Replication), the top replica stores A and B, whereas the bottom replicas store only one of the objects. There is no arrow between the bottom replicas because they do not store copies of the same objects (adapted from [83]).

Full replication, illustrated in fig. 2.6a, implies storing copies of all data items on every site. On the one hand, this approach ensures availability since every item is available on every site. On the other hand, it is resource-intensive as it requires every machine to store a full copy of the data [83, 67].

Partial replication, illustrated in fig. 2.6b, involves storing copies of only a subset of the data on each site. This approach is less resource-intensive than full replication since each site stores only a portion of the data. However, it may compromise availability since not all data items are available on every site [83, 67].

Overall, with partial replication, it is possible to reduce the cost of data storage and communication between servers [67] because updates are propagated to fewer replicas. Despite this, having fewer copies of the data may reduce fault tolerance and availability. Therefore, "in partial replication, it is very important to find a right replication factor" [83, p. 304].

2.1.3 Consistency

In the context of distributed systems, consistency refers to the degree to which nodes agree on the order of events. In replicated systems, ensuring consistency becomes challenging since it is necessary to propagate the changes to all the replicas when one of the copies is updated, which may penalize performance.

This trade-off motivated the study of a broad set of consistency models, ranging from linearizability [35], the strongest consistency model, to eventual consistency (EC) [95], the one that provides the weakest guarantees. Consistency models specify the conditions under which the replicas will converge to the same state and the guarantees provided by the replication protocol concerning the order of operations, the visibility of updates, and the handling of conflicts.

Linearizability "provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response" [35, p. 485]. A linearizable system behaves as if it was not replicated, thus simplifying programming and improving

understandability — "if the application program gets the same behavior as with a single site, unreplicated system, then writing the application logic is no different to conventional programming" [26, p. 4].

Nevertheless, according to the CAP theorem [32], it is infeasible to provide linearizability when aiming toward highly available and fault-tolerant systems, as a distributed data store can provide at most two of these three guarantees: consistency, availability, and partition tolerance; and one of the two must be partition tolerance. "Faced with the choice of at most two of these properties, many systems have chosen to sacrifice strong consistency to ensure availability and partition tolerance" [55, p. 2]. Furthermore, in a replicated system, one has to choose between latency and consistency in the absence of failures [2]. Thus, owing to the availability and performance benefits of weak consistency over stronger consistency models, "a common theme in replication research is to seek improved performance by giving up some level of consistency between replicas" [26, p. 2].

EC has emerged as an attractive form of weak consistency, especially with the recent rise of cloud computing services. With EC, "if no new updates are made to the object, eventually all accesses will return the last updated value" [95]. However, the lack of ordering guarantees of EC makes it less intuitive, as reads may not reflect the last updates, and different nodes may return incoherent values.

In order to avoid the potential anomalies of EC, researchers have proposed novel consistency models [3, 80] and explored ways to combine multiple models in the same system [50, 90].

2.1.3.1 Causal consistency

Causal consistency (CC) [3] was introduced in 1995, based on Lamport's concept of potential causality (*happened before*) [47], and motivated by the significant latency penalty entailed by strong consistency models to establish a common order of operations. On the one hand, its ability to surpass the limitations and anomalies of EC, mainly the lack of guarantees regarding the order of operations, makes it easier for developers to reason about the system's behavior. On the other hand, it minimizes the coordination overhead of strong consistency models, thus achieving lower latency. Due to these benefits, several academic studies have built causally consistent systems, some of which are reviewed in section 3.2.

This weak consistency model establishes a partial order of events that is consistent with the order defined by the *happened before* relationship [47]. Thus, all processes observe causally related operations in the same order, even though they may disagree on the order of concurrent operations. As stated by Brzezinski, Sobaniec, and Wawrzyniak [16], CC may be provided as the combination of the following session guarantees [89, p. 141] (quoting from the original paper):

Read-your-writes: read operations reflect previous writes.

Monotonic-reads: successive reads reflect a non-decreasing set of writes.

Write-follows-read: writes are propagated after reads on which they depend.

Monotonic-writes: writes are propagated after writes that logically precede them.

To illustrate the benefits of CC over EC, imagine the case of fig. 2.7, where Alice removes her coach from her friends. After that, Alice makes a post complaining about the game. With EC, due to the lack of ordering guarantees, the first operation might be executed in another replica later than the second request. Therefore, it would be possible for the coach to see Alice's post. On the other hand, if the system provides CC, as the operation that removes the coach from Alice's friend precedes the creation of the post, all replicas would apply the operations in the same order, avoiding this undesirable scenario.

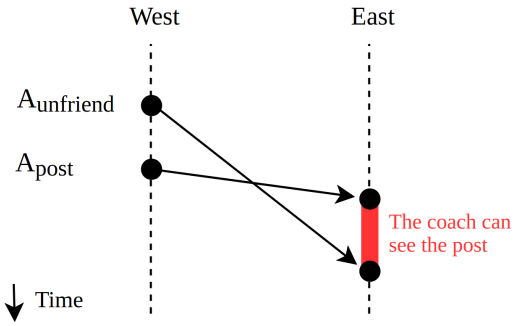


Figure 2.7: Possible anomaly introduced by Eventual consistency (EC). In the west coast data center, Alice removes her coach from her friends and then makes a post complaining about the game. However, in the east coast data center, the first operation is executed later than the second. Consequently, the coach can see Alice's post (adapted from [55]).

In practice, CC can be enforced using the mechanisms described in section 2.1.1.2, namely logical clocks [47], vector clocks [28, 64], and HLCs [44]. However, in some applications, "there is no need to register causality for all the events in a distributed computation" [10]. "For example, to modify replicas of data, it often suffices to register only those events that change replicas." [10] To this end, version vectors [74], whose usage is illustrated in Figure 2.8, are a mechanism similar to vector clocks but that only track the causality of the relevant events. Other alternatives, such as dotted version vectors [77], have been proposed to address the scalability limitations of classic version vectors.

2.1.3.2 Causal+ Consistency

CC does not impose an order between concurrent events. Therefore, in a replicated causally consistent system where, for example, key x is replicated in nodes A and B, if both nodes write concurrently to x, then they may diverge forever.

With this in mind, Lloyd *et al.* [53] introduced the definition of *Causal Consistency with convergent conflict handling*, which is also referred to as causal+ consistency (CC+). This consistency model extends CC to ensure that all replicas deal with conflicts identically and, thus, that they eventually converge to the same state after exchanging their write operations. One common

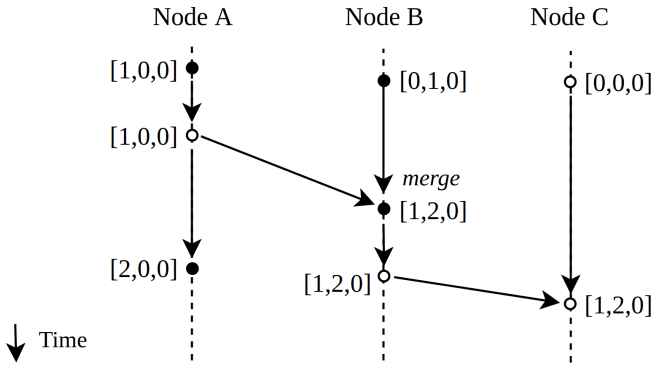


Figure 2.8: Example of the usage of version vectors to track the changes of a data item between replicas. A • represents an event registered for causality, whereas a ○ represents an event not registered for causality. An arrow between two events represents the *happened before* relationship. The vectors near each event represent the version vector of the node upon that event. Nodes A and B update the data item concurrently. Node A propagates its state to node B in a message. The receiver detects that the events are concurrent and merges the updates. When replica C receives the new state, it updates its version vector but does not create a new version because it has not performed any concurrent update (adapted from [10]).

way to ensure convergence is the last-writer-wins rule, with which, upon two conflicting writes, the one with the highest timestamp overwrites the other.

2.1.3.3 Transactions

CC alone does not overcome all the challenges of real-world workloads, where data does not fit on a single machine and must be partitioned over multiple servers [55]. TAO, for example, serves over ten billion requests per second on a changing data set of many petabytes [18], which makes it necessary to split the data across distributed nodes to scale capacity and throughput. With data spread across servers, the client's reads may reach different servers at different times, possibly resulting in an inconsistent view of the data.

To exemplify, assume the scenario above where Alice removes her coach from her friends and then makes a post complaining about the game. Even though causality assures that the order of the operations is preserved, the coach may still see the post in the scenario illustrated in fig. 2.9, where the friend verification and the post-fetching requests are sent to different servers. In that case, the friend verification request may arrive at a server before Alice unfriends the coach, and the post may be fetched from the other server after Alice removes the coach from her friends.

In order to avoid these potential anomalies and ensure a unified view of the data across partitions, previous research relies on read and write transactions.

A transaction is a unit of work commonly used in data storage systems to guarantee consistency in the presence of failures and ensure isolation. A transaction may comprise several read and write operations on multiple data items stored across partitions.

Transactions are defined by four key characteristics, which are commonly referred to as ACID properties [34]:

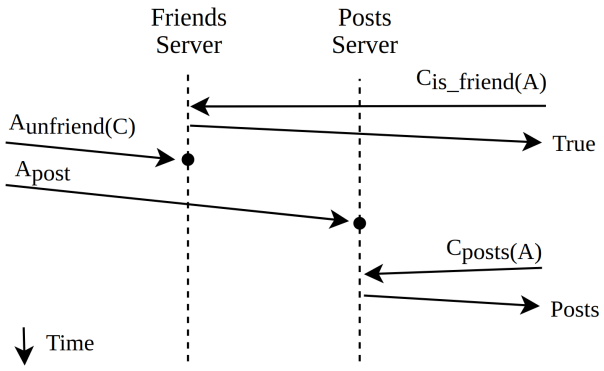


Figure 2.9: Possible scenario when using Causal consistency (CC). Alice removes her coach from her friends and then makes a post complaining about the game. However, different servers handle the post requests and the friend requests. Therefore, if the friend verification request arrives at the "Friends Server" before Alice unfriends the coach, and the posts are fetched from the "Posts Server" after Alice makes the new post, the coach may still be able to see the new private post (adapted from [55]).

Atomicity: "It must be of the all-or-nothing" [34, p. 289], so either all the operations in a transaction are executed, or none are.

Consistency: A transaction must take the system from one valid state to another. Hence it must preserve the consistency of the system.

Isolation: "Events within a transaction must be hidden from other transactions" [34, p. 290]. This ensures that the execution of one transaction does not affect the execution of other transactions.

Durability: Once a transaction has committed its results, they must survive subsequent failures.

Transactions that only perform read operations and do not modify the data are usually referred to as read-only transactions (ROTs) [31]. These transactions make it possible to extract a consistent view of data spread across different servers. For instance, using ROTs in the previous scenario, as shown in fig. 2.10, the coach would either extract a view where he is still friends with Alice and can see her posts or where he is no longer friends with Alice and cannot see her private posts.

By taking advantage of the knowledge that these transactions only read, many systems [53, 56, 54, 23, 25, 4, 19] use specialized algorithms to handle their processing.

Additionally, to enable atomic writes on multiple items (i.e., ensure that multiple writes are either all made visible or none), some works use write-only transactions (WOTs).

To enable even stronger semantics, namely to allow transactions that comprise both read and write operations, Akkoorath *et al.* [4] defined Transactional causal consistency (TCC), which extends CC+ with read-write transactions. TCC ensures the atomicity of write operations and that transactions can read from a causally consistent snapshot.

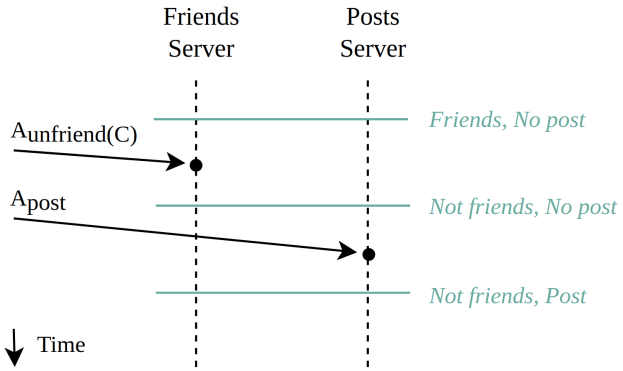


Figure 2.10: Causal consistency (CC) with ROTs. Alice removes her coach from her friends and then makes a post complaining about the game. In this scenario, the usage of CC and ROTs makes it possible to extract a consistent view of data spread across different servers: the coach would either extract a view where (1) he is friends with Alice and the incriminating post has not been published yet, (2) he is not Alice's friend and the incriminating post has not been published yet, or (3) he is not Alice's friend and the incriminating post was already posted, even though he cannot see it because it is private (adapted from [55]).

2.2 Read-Heavy Systems

The prevalence of reads is evident in several real-world applications [15, 18, 69, 92, 100]. Due to the pervasiveness of read requests, these applications have usually been designated as read-heavy or read-intensive. A well-known example of read-heavy applications is social networks. By way of illustration, Facebook reported TAO's ability to process "a billion reads and millions of writes each second" [15, p. 49], with 99.8% of the client requests being reads. A subsequent work [18, p. 1967] disclosed that TAO could support "over ten billion reads and tens of millions of writes per second on a changing dataset of many petabytes" and that reads accounted for 99.7% of Meta's social network requests workload. The dominance of reads is also evident in the case of Linkedin, where data "is written once, and read many times (>95% read traffic)" [69, p. 254]. Moreover, Linkedin's production environment serves up to "10K requests per second across more than 400 million users" [69, p. 253]. Even though social networks are the prevailing example in most studies [15, 18, 69], a diversity of distributed applications, such as e-commerce systems [100] or other content delivery systems whose focus is on delivering content to the user, like Wikipedia [92], are also prone to read-heavy workloads.

Due to the intrinsic read-heavy nature of these systems, they all share similar scalability, availability, and performance requirements. For instance, Noghabi *et al.* argue that: "A worldwide social network has to continually serve billions of variable-sized media objects" [69, p. 253], which "must be stored and served with low latency and high throughput by a system that is geo-distributed, highly scalable, and load-balanced" [69, p. 253]. Similarly, Bronson *et al.* [15, p. 49] assert: "The personalized experience of social applications comes from timely, efficient, and scalable access to ... the social graph".

Given the importance of read latency and throughput, recent research has explored novel techniques for increasing read efficiency in geo-replicated systems, yielding critical insights for the design of read-heavy systems. The aforementioned results will be highlighted in the following subsections.

2.2.1 SNOW Theorem

Preliminary work by Lu *et al.* [56] highlighted the importance of reducing the overhead of ROT algorithms, which are commonly used to ensure isolation and extract a consistent view across a partitioned data store. More importantly, this study proved the existence of a fundamental trade-off between the power and latency of ROT algorithms. More specifically, this work presents the *SNOW Theorem*, an impossibility result that shows that no ROT algorithm can simultaneously provide all the four desirable properties of ROTs (*SNOW properties*), namely:

- **Strict serializability (S)**, which guarantees that the transactions respect a total order compatible with their real-time ordering (linearizability [35]), take place atomically and are fully isolated (serializability), i.e., a transaction does not observe partial effects of other transactions.
- **Non-blocking operations (N)**, which ensures that each server can handle the operations within a ROT without blocking for any external event.
- **One response per read (O)**, which indicates that only one value is sent for each read (a property designated as *one version*); and that the client sends at most one request to each server and the server sends at most one response back (a property known as *one round*).
- **Write transactions that conflict (W)**, which denotes the ability of a ROT algorithm to coexist with conflicting write transactions.

The aforesaid properties are either associated with optimal latency or with high power. To be more precise, as stated by Lu *et al.* [56, p. 139]:

> *"Providing optimal latency requires providing the non-blocking (N) and one response (O) properties. Providing the highest power requires providing strict serializability (S) and being compatible with conflicting write transactions (W)".*

Considering the impossibility result of the *SNOW theorem*, Lu *et al.* define any ROT algorithm as *SNOW-optimal* if it achieves three of the four *SNOW properties*.

2.2.2 NOCS Theorem

Following the results from SNOW [56], Lu, Sen, and Lloyd [58] refined the latter definition of performance optimality for ROT algorithms by accounting for both ROTs' latency and throughput. More specifically, they defined the three performance-optimal properties of ROT algorithms (*NOC properties*):

Non-blocking: Each read request is processed without waiting for any external event.

One-round communication: The algorithm uses only one parallel round of on-path messages and no off-path messages.

Constant metadata: The amount of information required to coordinate consistent values in the system does not grow given the size of the system, transaction, or the number of concurrent operations.

As these properties are characteristic of simple reads, they capture the minor coordination overhead and the best performance that can be achieved with ROTs. Non-blocking transactions result in lower latency and higher throughput because transactions do not have to wait for locks, timeouts, or other events in order to complete and because it avoids context switches. Similarly, the *one-round* property enables lower latency and higher throughput because transactions require fewer network and CPU resources. Finally, fewer metadata leads to lower consumption of resources and hence better performance.

The *NOCS theorem* states that no ROT algorithm can be both performance-optimal and provide strict serializability, the strongest form of consistency. Thus, a system must either respect the three *NOC properties* or provide strict serializability.

2.3 Value Semantics

In computer science, the term *value* has been used to refer to an eternal, unchangeable, and implementation-agnostic abstraction that can neither be created nor destroyed and thus is atemporal and non-instantiable [61]. The immutability of *values* makes them sharable, conveyable, and perceivable [37].

Sequences of *values* can describe the evolution of the state of a concrete entity, i.e., "an individual thing that comes into and out of existence in space and time" [87, p. 1].

On the other hand, the term *semantics* has been used to refer to the system's behavior [33].

In the scope of programming languages, the term *value semantics* has been used to refer to a design principle that provides an immutable and persistent interface for the programmer, presenting data structures and types not subject to mutation post-construction [38]. For example, with *value semantics*, modifications perform a logical copy of the value instead of creating a new alias to the same memory address. Thus, modifying a copy does not affect the original value. Furthermore, equality comparisons only compare the contents (*values*) and not the identity [13]. Finally, *value semantics* enables referential transparency, i.e., "we may replace one expression with another of equal value anywhere in a program without changing the meaning of the program" [66]. Even though these characteristics are intrinsic to primitive types, most languages use *reference semantics* for objects and collections.

Overall, a programming language that supports *value semantics* effectively detaches state and identity (e.g., variable name), with control over each independently. In contrast, in languages that provide *reference semantics*, the notions of state and identity are tightly coupled [7] and thus cannot be treated independently.

In database systems like Spanner [19], Multi-Version Concurrency Control (MVCC) provides a way to realize this idea of causal value succession by keeping immutable versions of each identity that can be retrieved without impeding the system's progress. Furthermore, MVCC algorithms can avoid anomalies when executing transactions concurrently by returning slightly stale *values*. With MVCC, each write generates a new immutable version of the data item; hence a snapshot at a specific timestamp yields the value of the latest version that precedes that timestamp, ensuring write isolation.

Analogously, within distributed systems, *value semantics* closely relates to the idea that each node depicts a state machine [51, 47, 79] in which the application of an operation, coupled with its output, produces a new version of the state, identifiable by the operation number [51]. Hence, each operation generates an immutable and persistent *value* and moves the node's state forward in logical time.

Since no formal definition of *value semantics* has been found, in the present study, based on the definition of the terms *value* and *semantics*, *value semantics* is abstracted as an interface for computation that enables the perception of *values* in time. *Values* are immutable data items that causally succeed like a series of atomic states, each resulting from applying an operation to the previous one. They are stable and perceivable at a given point in time, and creating a *value* does not impede the perception of prior *values* [36, 37].

Achieving *value semantics* is desirable because *values* are semantically transparent, provide reproducible results, and are stable, thus simplifying decision-making [36, 37]. Moreover, by enabling developers to observe the progression of *values* in time, *value semantics* can also help identify any sequence of operations that resulted in unexpected behavior, making it easier to reproduce it and, thus, simplifying debugging.

2.4 Summary

This chapter outlined the fundamental concepts required to understand the present work.

Section 2.1 addressed relevant topics in the scope of distributed systems, such as physical and logical time (section 2.1.1), replication (section 2.1.2), and consistency (section 2.1.3).

Firstly, section 2.1.1 focused on the notion of time in distributed systems. Each node has its own physical clock and, thus, its independent perception of time. This lack of a global clock requires synchronization to ensure consistency. However, asynchrony and network unreliability make time uncertainty inevitable when using physical clocks. With this in mind, Lamport introduced the concept of logical time by defining the *happened before* relation [47], which establishes a partial order between events.

Section 2.1.2 defined replication, where a copy of the data is kept on several nodes, and its main benefits, namely reliability, availability, performance, and scalability. With full replication, a copy of the data is stored on every site, whereas with partial replication, only a subset of the data is stored on each site.

Section 2.1.3 described some consistency models, emphasizing CC. Consistency models range from linearizability [35], the stronger model, to weaker consistency models, such as CC [3] and EC [95], each providing different consistency guarantees within a distributed system. The incompatibility of stronger consistency models with high availability [32] and low latency [2] and the anomalies of EC make CC an attractive model. CC is based on the *happened before* relation and can be enforced using logical clocks [47], vector clocks [28, 64], HLCs [44], or version vectors [74]. CC+ extends CC with the guarantee that, eventually, all replicas will converge to the same state. To ensure a consistent view across shards, CC is usually coupled with ROTs.

Section 2.2 highlighted the prevalence of read-dominant workloads in many real-world applications, such as social networks [15, 18, 69], e-commerce systems [100], or other systems whose focus is on delivering content to the end user [92]. The read-heavy nature of these applications led recent works to focus on optimizing the performance of ROT algorithms. In this regard, SNOW [56] defined the properties that make ROTs latency-optimal and PORT [58] formalized the notion of performance-optimality for ROTs based on the characteristics of simple reads, which capture the minor coordination overhead: performance-optimal ROTs are non-blocking, take a single round of on-path communication, and use constant metadata.

Finally, section 2.3 introduced the concept of *value semantics* within the scope of distributed systems and its benefits in terms of auditability. In the context of this work, *value semantics* is abstracted as an interface for computation that enables the perception of values: immutable, stable, and perceivable data items that causally succeed like a series of atomic states, each resulting from applying an operation to the previous one.

Chapter 3

State of the Art

Contents

This chapter reviews the state of the art of causally consistent distributed systems. Section 3.1 describes the systematic literature review that guided the research process. Then, section 3.2 identifies the existing causally consistent systems with support for ROTs. In particular, it describes the implementation of each of the causally consistent systems identified in the literature review (section 3.2.1), discusses the design of these systems through the lens of the *NOC properties* of ROTs (section 3.2.2), and specifies the common strategies employed to ensure causality (section 3.2.3). Then, section 3.3 reviews the design choices of two causally consistent architectures, presenting relevant insights for the implementation of the proposed reference architecture. Section 3.4 identifies the common metrics used to evaluate causally consistent distributed systems. Finally, section 3.5 summarizes the main findings on the topics mentioned above.

3.1 Systematic Literature Review

A systematic literature review was performed to survey the state of the art in this dissertation's domain and to address some of its research questions, particularly **RQ1**, **RQ2**, and **RQ3**. Adopting this research process results in a reproducible analysis and reduces bias, thus leading to reliable and verifiable findings [82].

3.1.1 Methodology

The methodology used for the systematic review process consisted of the following steps:

(1) **Research Questions:** Identify the research questions that will drive the literature review.

(2) **Data sources:** Define the data sources to be used.

(3) **Base references:** Identify the base references on the dissertation's proposal bibliography and get acquainted with their core ideas.

(4) **Search Query:** Formulate a research query based on the research questions.

(5) **Inclusion and exclusion criteria:** Define the inclusion and exclusion criteria, particularly the time frame to be considered, the type and subject of the documents, and other criteria to include or exclude a result.

(6) **Screening for inclusion:** Identify relevant documents according to their similarity to the domain. Analyze the title, abstract, and keywords in the first phase and scan the document in the second phase.

(7) **Backward search:** Perform a backward search in the base references to identify related work that might be missing.

(8) **Expert advice:** Consult experts in the field to cross-check the search's completeness.

(9) **Full-text analysis:** Read the full text of the relevant documents and consult the corresponding references if new domain concepts arise.

The above process was iterative. Search results helped refine the research questions and search queries, thus increasing the results' relevance. In order to ensure a reproducible literature review, each step of the process was reported [72] in the subsections below and illustrated in the flow diagram [70] of fig. 3.1.

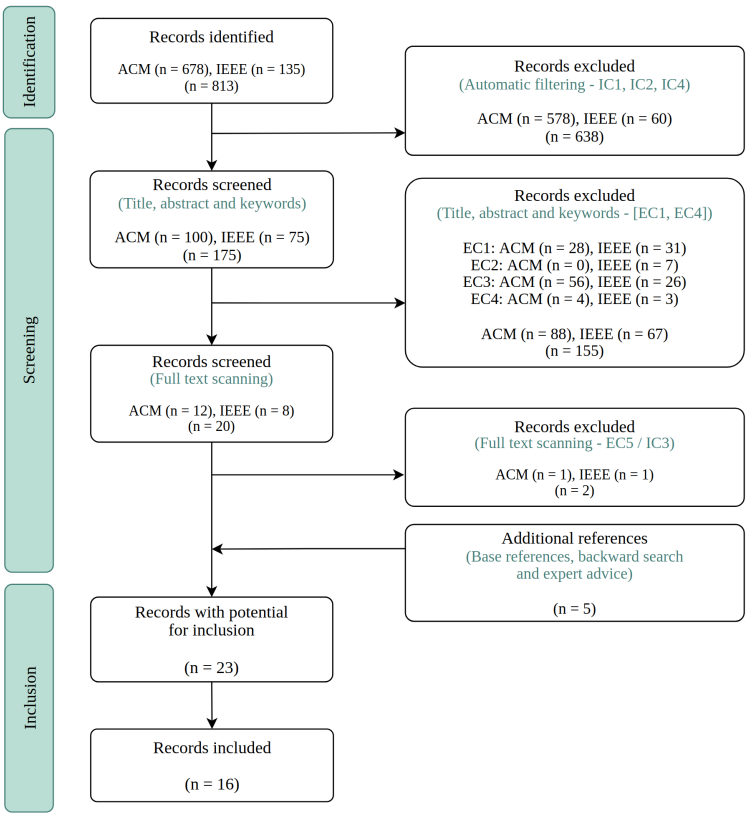


Figure 3.1: Systematic literature review.

3.1.2 Research Questions

The research questions addressed in this review were the following:

RQ1. What are the ideal properties of a geo-replicated causally consistent read-heavy system?
The prevalence of reads in real-world applications indicates the significance of optimizing the latency and throughput of ROTs for improving the system's overall performance and meeting customer expectations and service requirements. In this regard, reviewing the results achieved in PORT [58], particularly the performance-optimal properties of ROTs, is crucial for understanding how a read-heavy system can be designed to ensure the desired performance.

RQ2. What properties and strategies do existing causally consistent systems employ?
In order to build a reference architecture for read-heavy systems, it is crucial to review the architectural and algorithmic properties of existing systems, analyze their trade-offs, and determine how their strategies for ensuring CC affect the performance-optimal properties of ROTs identified in **RQ1**.

RQ3. What metrics have been used to evaluate these systems?
Given that any reference architecture must be empirically validated and evaluated to prove its applicability, it is relevant to identify which metrics must be considered when assessing the implementation of distributed causally consistent systems.

Table 3.1 maps these research questions to the sections where they are addressed.

ID	Research question	Related sections
RQ1	What are the ideal properties of a geo-replicated causally consistent read-heavy system?	Sections 2.2.2 and 3.2.2
RQ2	What properties and strategies do existing causally consistent systems employ?	Sections 3.2 and 3.3
RQ3	What metrics have been used to evaluate these systems?	Section 3.4

Table 3.1: The research questions addressed through the literature review and their corresponding sections in the document.

3.1.3 Data Sources

The digital libraries selected for the literature review process were ACM Digital Library and IEEE Xplore. The ACM Digital Library platform comprises a bibliographic database on the field of computing, namely the ACM Guide to Computing Literature, and IEEE Xplore provides access to highly-cited publications in the field of computer science, which makes them appropriate for the scope of the present work.

3.1.4 Base References

The base references of this dissertation portray relevant insights for optimizing read performance in causally consistent systems. SNOW [56] identifies the properties that make ROTs latency-optimal. PORT [58] formalizes the notion of performance-optimal ROTs not only through the lens of latency but also by considering throughput. Therefore, these works provide a way to analyze existing systems and identify room for improvement in their design, especially concerning read performance, which is crucial for read-heavy applications. Additionally, PaRiS [85] was the first system to provide CC and non-blocking ROTs in a partially replicated data platform.

3.1.5 Literature Analysis

3.1.5.1 Inclusion and Exclusion Criteria

In order to select the documents to be included, the systematic literature review followed the inclusion and exclusion criteria documented in table 3.2.

Inclusion Criteria	
IC1	Must be on the topic of distributed systems.
IC2	Published between 2011 and 2022.
IC3	Proposes a novel causally consistent distributed system that supports ROTs.
IC4	Research articles published on the proceedings of distinguished conferences or journals on topics associated with distributed systems.
Exclusion Criteria	
EC1	Presents a study unrelated to distributed systems.
EC2	Is not a research article published on the proceedings of a distinguished conference or journal on topics associated with distributed systems.
EC3	Presents techniques, consistency models, systems, or others, but not the actual implementation of a novel causally consistent distributed system.
EC4	Reviews the literature within the scope of distributed systems but does not present an actual implementation of a causally consistent distributed system.
EC5	Presents a system that does not support ROTs.

Table 3.2: Inclusion and exclusion criteria.

3.1.5.2 Search Queries

The initial queries were formulated based on the research questions. Listing 3.1 shows the query used in ACM Digital Library, whereas listing 3.2 illustrates the query used in IEEE Xplore. Considering that the included results would be assessed through the lens of the performance-optimal properties of ROTs (described in section 2.2.2), these search queries were refined to identify causally consistent systems that support ROTs.

```
ACM (ACM Full-Text Collection + ACM Guide to Computing Literature)
[[All: "distributed"] OR [All: "geo-replicated"]] AND
[All: "transactions"] AND [All: read*] AND [All: "evaluation"] AND
[[All: "causally consistent"] OR [All: "causal consistency"] OR
[All: causality]] AND
[[Abstract: "distributed"] OR [Abstract: "geo-replicated"] OR
[Abstract: "causally consistent"] OR [Abstract: "causal consistency"] OR
[Abstract: causality]]
```

Listing 3.1: The search query used in ACM Digital Library

```
(("Full Text Only": "distributed" OR "Full Text Only": "geo-replicated") AND
"Full Text Only": "transactions" AND "Full Text Only": "read" AND "Full Text
    Only": "evaluation" AND
("Full Text Only": "causally consistent" OR "Full Text Only":"causality" OR
"Full Text Only": "causal consistency") AND
("Abstract": "distributed" OR "Abstract": "geo-replicated" OR
"Abstract": "causally consistent" OR "Abstract":"causal consistency" OR "
    Abstract":"causality"))
```

Listing 3.2: The search query used in IEEE

3.1.5.3 Review Process

The review process outlined in fig. 3.1 can be described by the following stages:

Identification:

The results were retrieved from each digital library using the queries described in section 3.1.5.2.

Screening:

(1) **Automatic filtering:** The results were refined using the available filtering options of each digital library. In ACM, the results were filtered according to **IC1** by restricting the ACM Computing Classification System to the following categories: Distributed computing methodologies, Distributed computing models, Distributed systems organizing principles, or Distributed architectures. In IEEE, **IC4** was applied by constraining the results to proceedings and journals. In both digital libraries, the results were filtered according to the publication date (**IC2**), only keeping the documents published between 2011 and 2022.

(2) **Filtering by title, abstract, and keywords:** The results were reviewed according to their title, abstract, and keywords. Documents presenting studies unrelated to distributed systems (**EC1**), not published in conferences or journals related to the topic

(**EC2/IC4**), that did not present implementations of causally consistent distributed systems (**EC3**), or that only reviewed the literature (**EC4**) were excluded.

(3) **Filtering based on full-text scanning** The results were filtered according to **EC5** and **IC3** to exclude documents that proposed causally consistent systems without support for ROTs.

Inclusion:

(1) **Additional references**: The missing base references (SNOW [56], and PORT [58]) and the results from the backward search (Contrarian [23]) were also included for review. Finally, through expert advice, two other publications (Bolt-on [8] and MongoDB [91]) were considered for review. Even though these works do not support ROTs, they present general architectures for causally consistent systems. Therefore, they present great relevance for the main goal of this work — providing a causally consistent reference architecture for causally consistent read-heavy systems.

(2) **Included publications:** Even though the literature review yielded twenty-three relevant results (documented in appendix A), it was necessary to select which would be included in the qualitative synthesis due to time constraints. Apart from the base references of the dissertation (SNOW [56], PORT [58] and PaRiS [85]), and given the importance of understanding the core ideas on top of which several systems were built, the selection for full-text analysis was based on the original publication date of the works, prioritizing the ones published prior to PORT [58], and on the number of citations. The final selection comprises the following publications: COPS [53], Eiger [54], ChainReaction [5], Orbe [24], GentleRain [25], SwiftCloud [98], Cure [4], Occult [65], COPS-SNOW [56], POCC [86], Wren [84], Contrarian [23], PaRiS [85], Eiger-PORT [58], Bolt-on [8], and MongoDB [91].

3.2 Causally Consistent Systems

CC has been used in geo-replicated systems due to its ability to surpass the ordering anomalies of EC while avoiding the latency penalty and availability constraints of stronger consistency models. Moreover, unlike EC, CC's intuitive semantics [23] makes it easier to reason about the programs' decisions.

Regardless of its benefits, CC alone does not overcome all the challenges of real-world workloads where data does not fit on a single machine. With this in mind, TCC extends CC by providing transactions that observe a causally consistent view of the data.

There are several trade-offs to consider when designing these systems. In particular, to optimize the latency and throughput of ROTs, it may be necessary to return slightly stale data to the user.

In the scope of read-heavy systems, it is relevant to analyze how different strategies and design choices affect the properties of the system and namely of read operations such as ROTs. To

that end, section 3.2.1 reviews the implementation of existing causally consistent systems, section 3.2.2 analyzes these systems through the lens of the performance-optimal properties of ROTs, and finally, section 3.2.3 analyzes how the strategies commonly used to enforce causality impact the systems' properties.

3.2.1 Existing Systems

This section reviews the architecture and algorithms of existing causally consistent systems to understand how they were implemented, the trade-offs in their design, and how they compare with other solutions.

3.2.1.1 COPS

Lloyd *et al.* [53] presented COPS, the first system to provide CC in a partitioned fully replicated key-value data store — previous designs [75, 12, 11, 45] assumed replicas were limited to a single machine and relied on log exchange to provide consistency, thus inhibiting scalability. COPS and COPS-GT, which extends the core version with non-blocking ROTs, not only realize CC+, using by default the last-writer-wins rule but also assure availability, low latency, partition tolerance, and scalability (also known as ALPS properties).

Each data center (DC) comprises a local copy of a linearizable [35] key-value store and a client library. This option ensures low latency for client operations and availability in the face of external partitions. Each key-value pair is associated with a version number and a list of dependencies. The client library supports read and write requests and, in the case of COPS-GT, ROTs. Moreover, it keeps track of the causal dependencies within each client's operations.

Every key stored in COPS has one primary node in each cluster (also named "equivalent node"), which handles read operations. Write operations are first committed locally, with the version assigned by the primary node (a Lamport timestamp [47]). Replication to remote nodes is handled asynchronously, and updates only commit remotely when the value's dependencies are satisfied. Moreover, to provide fault tolerance, COPS relies on chain replication [93].

COPS-GT also provides non-blocking ROTs, which take at most two rounds within the DC. In the first round, the system retrieves from the respective primary node the latest version of each key in the transaction, along with their dependencies. If some dependencies are unsatisfied, the library issues a second round to request the newest version of the respective keys seen in any dependency list from the first round.

Both systems garbage-collect client metadata to minimize storage overhead. COPS-GT also garbage-collects versions and dependencies. COPS only verifies the nearest dependencies, reducing the number of dependency checks.

Comparing both systems demonstrates that COPS-GT's throughput becomes competitive with COPS for read-heavy workloads.

3.2.1.2 Eiger

Lloyd *et al.* introduced COPS' [53] successor, Eiger [54]. Besides providing low latency and CC with ROTs, Eiger uses a rich column-family model and supports WOTs. Like COPS, it also assumes full data replication across DCs.

To provide CC, Eiger ensures that all its dependencies are applied before an operation is performed. However, in contrast to COPS, which stores dependencies on values, Eiger tracks dependencies on operations, which is crucial to improve efficiency in the column family data model, where many columns can be simultaneously written or read for a single key. Furthermore, Eiger tracks only the one-hop dependencies for each write at the client, minimizing dependency check overhead.

Similarly to COPS, Eiger's ROT algorithm usually completes in a single round of local reads and two rounds in the case of concurrent updates. However, it achieves this using logical clocks and timestamps instead of explicit dependencies. This way, Eiger achieves greater efficiency and is more tolerant under long partitions between DCs, while COPS may suffer a metadata explosion that can degrade availability.

Eiger also proposes a novel WOT algorithm that atomically updates a set of keys across different servers of a DC. This algorithm is lock-free and does not block concurrent ROTs, even though it may affect the number of rounds a ROT takes to complete. For instance, if a ROT is issued while a WOT is pending, then the client library must request a second version at a specific logical time. If there are still pending WOTs executing upon the second round, then the third round of communication is needed to resolve the ordering.

Eiger's experimental evaluation demonstrates that the overhead of providing CC with ROTs and WOTs is low compared to a non-transactional, eventually consistent baseline. Furthermore, compared with COPS-GT, Eiger's ROTs achieve higher performance, as its implementation avoids the metadata explosion that COPS-GT's ROTs can suffer.

3.2.1.3 ChainReaction

Similarly to COPS [53], ChainReaction [5] is a key-value store that delivers CC+. Its architecture encompasses a set of DCs, each comprising several client proxies and data servers. Client proxies receive requests and redirect them to the appropriate data servers, which in turn handle the requests for a set of keys defined through consistent hashing [41]. To ensure fault tolerance and distribute load across nodes, these servers employ a novel variant of the chain replication [93] protocol. In this variant, writes may return as soon as the first k replicas process them (where k is the fault tolerance of the chain), and the remaining propagation is lazily handled.

ChainReaction's client library provides an interface for `PUT`, `GET`, and `GET-TRANSACTION` operations and also manages the client's metadata. In particular, it keeps $\langle key, versionVector, chainIndexVector \rangle$ tuples for each object accessed by the client, where the version vector identifies the version of the object and the $chainIndexVector$ keeps an estimate of how far the current version has been propagated across chains in all DCs.

Before handling a PUT, the protocol guarantees that the local DC has applied all the PUT dependencies of the current request. For that purpose, it directs a blocking read to the tail of the chains of the objects belonging to the client's causal history (referred to as *dependency stabilization procedure*). Then, the request is forwarded to the appropriate server head and passed along the chain until it reaches the k^{th} element. The client receives the response and sets its local entry of the *chainIndexVector* to k. When the lazy propagation completes, a notification is sent up the chain and to sibling chains in remote DCs. In the background, the request is sent to a *remote-proxy* which combines several updates in a single *remote-update* and propagates it to other DCs. *Remote-proxies* rely on version vectors to ensure causality. Moreover, to ensure the order of operations within a *remote-update*, Almeida *et al.* utilize Adaptable Bloom Filters [20]. Eventually, an update becomes stable in all DCs. In the case of concurrent updates, the protocol uses the last-writer-wins policy.

For GET requests, the client proxy consults the i^{th} entry of the *chainIndexVector* to decide which data server must handle the operation — it selects a data server at random with an index from 0 to $chainIndexVector[i]$, thus distributing the load between servers. As updates are propagated asynchronously, the local chain may be unable to respond to a GET. In this case, the server must redirect the request to another DC or block the operation until its value can be returned.

GET-TRANSACTIONS are assured in each DC by a sequencer process that orders the operations through sequence numbers. When a GET-TRANSACTION is issued, the server returns the version created by the last PUT that preceded the transaction. The transaction is aborted if the required version is unavailable for more than a certain timeout. Furthermore, it may be the case that one of the requested values includes dependencies on operations performed on other DCs. In that case, the algorithm consists of two rounds: a blocking read to get the values of the missing dependencies and the GET-TRANSACTION.

On read-heavy workloads, evaluating ChainReaction against a system that provides stronger local guarantees like COPS reflects better throughput and performance. Additionally, in COPS, dependencies make up most of the message payload, and the message's size increases during execution, resulting in higher communication overhead. In contrast, ChainReaction's metadata is limited to a key, value, and two bloom filters, and the messages' size is constant.

3.2.1.4 Orbe

To provide scalable CC in replicated and partitioned data stores, Du *et al.* implemented Orbe [24], a distributed key-value store whose functioning primarily relies on dependency matrices and physical clocks.

Like previous systems, Orbe assumes a fully replicated data store with M DCs and N partitions per DC. A client only interacts with his local DC servers.

The key novelty of Orbe is the usage of dependency matrices. Each row of the matrix is a version vector that stores the causal dependencies of each replica of a partition, more specifically, the last update timestamp it has seen so far from that replica.

Orbe supports read (`GET`) and write (`PUT`) operations, as well as ROTs (`GET-TX`). Clients keep track of their causal history by storing the nearest dependencies of the current session in a dependency matrix DM_c with N rows and M columns. Furthermore, clients maintain a *physical dependency time* (PDT_c) for the current session, which stores the greatest physical update timestamp of all its dependencies.

Partitions keep a version vector (VV) to track the updates applied by each of its replicas and the total of updates it executed locally. They also keep a physical version vector (PVV) that tracks the last physical timestamp seen from each replica of that partition. If replicas do not exchange updates for some time, the periodic broadcast of heartbeat messages ensures that these values are kept up to date.

`GET` operations are sent to the correspondent partition of the local DC. Its reply includes the update timestamp of the requested item and the physical update timestamp, which are used to update DM_c and PDT_c, respectively.

Upon a `PUT` operation, the client attaches DM_c and PDT_c to the request. To ensure causality, the partition that receives the request waits until its physical time is higher than PDT_c and, only then, increments its VV and uses it to timestamp the new item. To support ROTs, items are also versioned with physical update timestamps using the physical clock value. Furthermore, it attaches DM_c and the replica id to the item. The reply, which includes the replica id and both update timestamps, is sent to the client. The client resets DM_c and stores the timestamp received. Furthermore, it updates his physical dependency time to the maximum of the current value and the item's physical update timestamp. Each partition replicates updates in timestamp order to other replicas. To guarantee CC, before applying the update of item d, partition n waits until it has installed the dependencies of d, (i.e., $VV \geq DM_d[n]$). It also assures that the other local partitions have installed all dependencies through dependency check messages. Finally, it updates VV and PVV.

ROTs are timestamped with a physical time slightly older than the actual clock time of the server to reduce the probability of a transaction being delayed and the duration of the delay. If the partition that performs the transaction does not store all the keys, then it retrieves them from other local partitions. However, before reading an item at a partition, that partition must have executed all local updates and applied all remote updates with update timestamps lower than the transaction's timestamp. Therefore, it waits until the timestamp is lower than the partition's clock value and lower than its minimum PVV.

If remote replicas or the network among partitions fails, the transaction may be delayed without knowing if all dependencies are satisfied. To solve this problem, Du *et al.* propose two possible approaches: re-execute the transaction with a smaller timestamp to avoid blocking on the same failing partition or use a two-round protocol like Eiger [54] to avoid returning a stale snapshot.

In addition, the authors propose an optimization they call *dependency cleaning*, which can be used to reduce the size of dependency metadata at the cost of higher communication overhead.

In comparison to COPS [53], Orbe delivers greater throughput since COPS must track all dependencies, whereas Orbe only tracks the nearest dependencies. Furthermore, Orbe uses sparse

matrix encoding to compress the dependency matrix and resets the matrix of a client after each update operation, which also minimizes throughput.

3.2.1.5 GentleRain

Du, Jiaqing, *et al.* proposed GentleRain [25], a causally consistent geo-replicated data store that, in contrast to previous implementations [53, 54, 5, 24], does not require dependency check messages to track causality. Instead, it uses physical clocks to generate scalar timestamps for each update, which leads to an increase in throughput and a decrease in storage and communication overhead. However, this strategy results in a slight staleness increase at remote DCs.

Their work assumes a multi-versioned data store where the data is split across partitions of a DC and fully replicated in other DCs. Each server has a loosely synchronized physical clock that provides monotonically increasing timestamps. The data store supports simple `GET` and `PUT` operations, snapshots, and ROTs.

Like in COPS [53] and Eiger [54], in GentleRain, updates become instantly visible locally but are propagated asynchronously to remote replicas. Each server resorts to a version vector to keep the last update seen from each replica, including itself. From it, each server computes its local stable time (*LST*), the minimum element of its version vector, which represents the highest update timestamp seen from all replicas. The global stable time (*GST*) is computed periodically and represents the minimum *LST* of all partitions within the same DC. Updates originating at remote DCs only become visible when their update timestamp is smaller than the *GST*.

In opposition to the snapshot operation, where the datastore may return a snapshot from any point in the past, ROTs must include any values the client has already read. Thus, the implementation either blocks the transaction until the client's time meets the *GST* or uses Eiger's protocol, which may take two rounds to complete.

The comparison between GentleRain and COPS shows that GentleRain yields a considerably higher throughput for read-heavy workloads, as COPS needs to track more dependencies.

3.2.1.6 SwiftCloud

Targeting client-side applications, which typically need to ensure high availability, Zawirski *et al.* proposed SwiftCloud [98], a distributed object database that scales with the number of client-side replicas and provides CC+ even in the presence of DC failures. Furthermore, SwiftCloud uses Conflict-Free Replicated Data Types (CRDTs) [80] for confluence, supporting high-level objects.

SwiftCoud's cloud infrastructure connects tens of geo-replicated DCs, each storing a full replica of the database, to thousands of partial client-side replicas, which specify the subset of the database they are interested in (i.e., their *interest set*). Clients read locally from a causally consistent cache and register updates in a log, which is asynchronously transferred to a DC and regularly pruned through checkpointing. The *interest set* of the client can change dynamically.

Each DC keeps the object versions in stable storage and a version vector (VV_{DC}) representing a recent version of the database. Clients keep a base version of the database provided by a DC and

its corresponding version vector, both provided by a DC through a notification protocol, and a log with the updates performed on the client side (U_C).

A client starts a transaction by setting its *snapshot vector* with the version vector of the base version it knows (VV_{DC}). Any internal updates performed during the transaction are stored in the transaction's buffer. Reads are performed from the snapshot vector merged with the internal updates of the client. The transaction is committed locally by assigning a client timestamp, a sequence number representing the number of writes performed by the client. Each update belonging to the transaction is added to the client's *commit log* together with its dependencies, which correspond to the transaction's snapshot vector.

Asynchronously, unacknowledged updates are transferred to the current DC in order. When a DC receives an update, it verifies if its dependencies are satisfied and, if so, assigns it a DC timestamp, stores it, updates its version vector, and acknowledges the client.

To tolerate DC failures, SwiftCloud's clients only depend on their internal updates and external updates that are stable in at least k DCs. Thus, the base versions transmitted from DCs to clients have been acknowledged by $k-1$ other DCs. The parameter k provides a trade-off between freshness and availability - a higher k means higher availability and lower freshness and vice versa. Considering this strategy, the client retries with another DC if a transfer request times out. On the other hand, if the DC is missing any external dependency, the client must retry with another. It is also possible that a DC is missing an internal dependency, in which case the client must resend possibly missing updates.

New updates are replicated to other DCs using uniform reliable broadcast. Upon receiving an update, a DC buffers it until all its dependencies are satisfied, stores it, and updates its version vector with the DCs timestamps of the update.

SwiftCloud was evaluated against the standard read-heavy and update-heavy Yahoo! Cloud Serving Benchmark (YCSB)'s workloads and through a social network workload that employs higher-level operations. The results suggest that the latency of reads and writes of cached operations is less than 1 ms, the throughput scales with the number of DCs, and that the metadata grows linearly concerning the number of DCs, all at the cost of a slight staleness under 1%.

3.2.1.7 Cure

In pursuit of a flexible programming model and stronger semantics, Akkoorath *et al.* introduced Cure [4], the first system to achieve TCC, which extends CC+ with transactions that can interleave read and write operations. Furthermore, Cure's implementation leverages CRDTs [80] to ensure the replica's convergence while providing a richer interface to developers.

Like previous systems, this work assumes a fully replicated multi-versioned key-value store where objects are replicated across M DCs, each comprised of N partitions. Cure's programming interface provides operations to start a transaction, read or update a set of keys, and commit or abort the transaction.

Each partition is equipped with a loosely synchronized physical clock that generates monotonically increasing timestamps, which are used to annotate the updates with their commit time.

Additionally, a vector clock with one entry per DC (*pvc*) keeps track of the last update timestamp received from each remote replica. Another vector clock (*GSS*) is used to track the timestamp of the last snapshot known to be available at all the partitions of the DC (i.e., the *globally stable snapshot*). To compute the *GSS*, partitions within each DC periodically exchange their *pvc* through a stabilization protocol. The minimum of the received *pvc* determines the *GSS*.

To start a transaction, the client sends his last seen update (*cvc*) to a local DC that acts as the transaction coordinator. First, the coordinator may need to wait until its *GSS* catches up with *cvc*. Afterward, it computes the snapshot visible to that transaction (*svc*) by copying the *GSS* and setting the local entry to the maximum of the *cvc* and the server's physical clock. The id of the transaction is returned to the client, who may then issue updates and read operations to the coordinator.

Each read request is forwarded to the responsible local partition together with the transaction's snapshot (*svc*). A server may need to wait until its vector clock (*pvc*) catches up with the transaction's snapshot (*svc*) before returning the latest version of the requested item with a timestamp at most equivalent to the requested snapshot. To ensure read-your-writes, the coordinator applies any updates performed to that same key in the current transaction. The result is returned to the client.

Upon receiving a commit request, the server performs a Two-Phase Commit (2PC) protocol. First, it sends a *prepare* message to the involved servers with the writes to be performed. Each server waits, if necessary, until its clock catches up with *svc* and then registers the write set in its log, records the transaction in its list of prepared transactions, and proposes its clock values as commit time. The coordinator chooses the maximum of the received timestamps and generates the commit vector clock of the transaction (*ct*) by applying this timestamp to the local DC's index on the *svc*. In the commit phase, the coordinator sends *ct* to the involved servers, which update their log, remove the record from the prepared list and add it with the commit time to the committed list.

The committed transactions with commit timestamps lower than all the prepared transactions are periodically propagated to replicas in other DCs. In the absence of updates, heartbeat messages are used for synchronization. When receiving a remote update, the replica applies the operation to its log and updates the index of the sender in the *pvc*, indicating that all the updates from that server until the received timestamp have been received. The visibility of the remote updates depends on the stabilization protocol described above.

Periodically, servers exchange the oldest snapshot vector clock of their ongoing transactions, compute the aggregate minimum, and prune their logs accordingly.

In terms of throughput, Cure outperforms Eiger [54] as the update rate increases due to the overhead incurred by the dependency checks. Furthermore, by employing vector clocks instead of scalar timestamps to track the stable snapshot, Cure achieves lower update visibility latency at remote sites than GentleRain [25], which also relies on a stabilization protocol. Additionally, it tolerates network partitions and DC failures. Nevertheless, these enhancements come at the expense of increased metadata size.

3.2.1.8 COPS-SNOW

Based on the *SNOW Theorem*, Lu *et al.* [56] proposed a design for COPS [53] (the predecessor of Eiger [54]) that makes it latency-optimal - "it keeps the non-blocking property of the current algorithm and adds the one response property" [56, p. 142]. To achieve this improvement, the trade-off made by COPS-SNOW is to incur extra costs on writes instead of reads to make reads more efficient, which is crucial in read-heavy workloads. In COPS, ROTs may complete in one or two rounds. Servers respond to the first request with the current value for the requested keys and each value's causal dependencies. The client must verify if those dependencies are satisfied and, in case they are not, send the second request for each of the values whose dependencies were not satisfied. As the second round only occurs if keys involved in the ROT are updated during the first round, COPS-SNOW flips this scenario by moving this verification to the writes. A write that is issued during a ROT must check if any of its causal dependencies have not been observed by an ongoing ROT (a technique referred to as *readers check* [23]). The current write should not be observed if any dependencies are not applied yet. Apart from the dependency checks, writes must also record the ROTs that observed the value they overwrote.

The evaluation of the proposed algorithm for varying write frequencies shows an improvement in latency for ROTs at the cost of lower system throughput when compared with COPS. This loss results from the additional messages in the write algorithm. Furthermore, COPS-SNOW's latency advantage increases as ROTs increase in size (i.e., number of keys read) because the ROTs in COPS are more likely to go to the second round. In the same scenario, however, the throughput of COPS-SNOW becomes worse relative to COPS because each write has more causal dependencies to check.

3.2.1.9 Occult

Motivated by the susceptibility of previous work against slowdown cascades, a problem that may be at the root of the industry's unwillingness to apply CC, Mehdi *et al.* introduced Occult [65], the first causally consistent system to work around this problem.

In previous causally consistent systems such as COPS [53], Eiger [54], and GentleRain [25], a shard in a replica only applies a write when all the shards in that replica have applied the write's causal dependencies. Therefore, if a shard fails, the visibility of the writes is delayed, affecting not only the faulty shard but impacting the performance of all the others (i.e., leading to a slowdown cascade).

To overcome this problem, Occult shifts the enforcement of CC from the data store to the clients. By adopting this optimistic approach, where writes are immediately applied in the data store, Occult trades off the availability of read operations from lagging shards for write visibility. Furthermore, it requires clients to track the dependencies concerning the entire data store.

Occult is a fully replicated multi-versioned key-value store with M DCs, split into N partitions. It implements a primary-replica architecture with one primary node per shard accepting writes and asynchronously replicating them to replica nodes.

To track causality, Occult proposes the use of shardstamps and causal timestamps. Shardstamps count the number of writes that a shard has accepted. Causal timestamps are vectors of shardstamps, each entry concerning a shard. They encode the most recent state observed by the client and the causal history of each write.

To compress the size of causal timestamps and avoid metadata overhead before large numbers of shards, the authors propose several methods to compress causal timestamps: *structural compression*, *temporal compression*, and *DC-isolation*. In the first strategy, *structural compression*, shards whose ids are congruent modulo n, where n is the number of entries of the causal timestamp, are mapped to the same entry. This strategy, however, results in false dependencies, which may affect read latency. To minimize this effect, shardstamps can be replaced by physical timestamps. Nonetheless, this strategy falls short when shardstamps of shards that map to the same entry are very different. In this regard, the authors propose *temporal compression*, a technique based on the intuition that recent shardstamps are more likely to generate false dependencies. In this strategy, $n-1$ entries are reserved for the most recent shardstamps, whereas a *catch-all* entry compresses the remaining shardstamps. Finally, *DC-isolation* consists of using a vector of causal timestamps with one entry per DC, an optimization that aims to reduce read staleness differences across DCs, which is affected by the existence of primary and secondary partitions on different DCs.

In Occult's protocol, clients typically issue requests to shards of the replica they are co-located with, even though they are not required to be sticky to that replica. A client library enforces CC and attaches the necessary metadata.

To perform a write, the client library attaches the client's causal timestamp to the request and sends it to the primary of the respective shard. The primary node increments its shardstamp and updates its entry of the causal timestamp to reflect the write, storing it together with the new value. Replication occurs asynchronously and in order to replicas. The response to the client library includes the causal timestamp of the write, which the client uses to update his own.

Read requests are usually sent to the local shard. The reply includes the most recent value for the requested key, its dependencies in the form of a causal timestamp, and the shardstamp of the contacted shard. The client verifies if the contacted shard entry matches the received shardstamp to know if it is safe to read the returned value. If it is not safe to read, the client can either retry until the local replica advances or forward the request to the primary shard, which invariably stores the most recent version. The response to the client library includes the requested values and the server's causal timestamp, which is used to update the causal timestamp of the client.

Occult's transactions protocol consists of three phases. During the read phase, the client retrieves the values for the requested keys from his local shards, forming his *read set*, and buffers the writes. During the validation phase, it first validates if the values form a causally consistent snapshot. Specifically, it checks if the items are pairwise consistent by verifying if a shardstamp of item a is at least as recent as its entry in the causal timestamp of item b and vice versa. If this check does not hold, the transaction aborts. Otherwise, the client tries to lock the access to the items of his *write set* by contacting the corresponding primary shards. In the absence of an

acknowledgment, the client retries and may eventually abort the transaction. If the request is successful, the server returns the causal timestamp of the item and the shardstamp of the new write, which is stored on the *overwrite set*. In the last step of the validation phase, the client verifies if its *read set* is at least as recent as its *overwrite set* (i.e., that the dependencies of the new transaction are present), aborting otherwise. Finally, in the commit phase, the client generates the commit timestamp of the transaction by combining the causal timestamp of the read snapshot with the received shardstamps. All writes are assigned the same commit timestamp in their primary shards, ensuring atomicity.

Occult was evaluated against an eventually consistent data store to understand the overhead introduced by CC. In contrast to previous systems, which typically measure throughput, Occult measures goodput instead, which does not count operations that abort nor retries. In a read-heavy workload with 95% reads, results from the evaluation suggest that *DC-isolation* is competitive with the eventually consistent system. Furthermore, they emphasize the trade-off between the causal timestamps' size and the reads' staleness.

3.2.1.10 POCC

Spirovska, Didona, and Zwaenepoel [86], backed by the fact that updates are often inherently replicated consistently [9, 59], and that network partitions and DC failures are infrequent [9, 14], argue that previous approaches to CC are overly pessimistic. In this regard, the authors propose Optimistic Causal Consistency (OCC), an optimistic protocol that trades off availability for higher data freshness, always returning the most recent available version to the client, even if its dependencies have not yet been installed.

On the one end, OCC does not rely on stabilization or dependency checking, thus imposing lower communication and computational overhead. On the other hand, it presents a blocking behavior because reads may need to wait for missing dependencies to be installed, making the system vulnerable to network partitions and DC failures. Therefore, under failure scenarios, the system must fall back to a pessimistic protocol, such as Cure's [4].

This optimistic strategy requires clients to store and transmit their dependencies when executing an operation. When performing a read, the client's history will determine if the server must wait for missing dependencies, and when writing, the new version of the item will take the client's history as its dependencies.

The authors implemented OCC on POCC, a multi-versioned key-value store whose data set is split into N disjoint partitions. Each partition is replicated at M different DCs. Clients connect to a node in the closest DC, which takes the coordinator role and forwards any requests for keys it does not store.

Each server relies on a loosely synchronized physical clock that assigns monotonically increasing timestamps to new writes. Furthermore, it keeps a version vector with one entry per DC (VV), which stores the highest update timestamp seen from each replica. Apart from the key, value, and creation timestamp, each item is stored together with its dependencies, represented by a dependency vector (DV) with one entry per DC.

To resolve dependencies upon a read, clients also hold a dependency vector (*DV*) and a read dependency vector (*RDV*), which represents the potential dependencies of the items he has read and is the maximum of the dependency vectors of those items.

POCC provides clients with the following operations: simple reads, writes, and ROTs.

A read returns the most recent version of the key compatible with the client's history. The client attaches its *RDV* to the request to perform a read operation. The server checks if its version vector is entry-wise greater than the client's *RDV*. If it is, it returns the latest version of the requested item. Otherwise, it must block until that condition holds. The server sends back the item, and the client uses the item's dependency vector to update its own *RDV* and *DV* and uses the update timestamp of the item to update *DV*.

In a write request, the client sends its *DV*, and the server must wait until its clock is higher than the maximum of *DV*, ensuring the new update gets a higher timestamp than its potential dependencies. Then, the server uses its physical clock value to set the timestamp of the new item, adds it to the corresponding version chain, and sends the reply with the update time, which the client uses to update *DV*.

In ROTs, the server defines the snapshot visible to the transaction as the items currently received by nodes in the local data center, whereas in other systems, the snapshots assigned to ROTs only include stable items. When performing a ROT, the client attaches its *RDV*. The server first determines the timestamp vector that defines the snapshot visible to the transaction (*TV*) - the entry-wise maximum between the version vector of the server and the client's *RDV*. The server sends that vector to all involved partitions. Each partition waits until its version vector is higher than *TV* and then returns the freshest version of that snapshot. All the items are then sent back to the client.

Replication is handled asynchronously to other DCs in timestamp order. A server that receives a remote update must insert the item in its version chain and update the corresponding entry of its version vector. Additionally, to allow items to be garbage collected, a partition that does not receive requests for local updates for some time broadcasts its latest clock time to its replicas through a heartbeat message.

When a network partition occurs, the session is reinitialized using a pessimist protocol based on Cure. The consequence of this is similar to what happens in other pessimistic approaches, where the fall-back server may not know the same versions as the old one.

POCC's performance was assessed against a pessimistic system, namely Cure*, a version of Cure that also supports simple read and write operations. In a transactional workload, the throughput varies identically with the number of contacted partitions in both systems, though POCC's throughput outreaches Cure's up to 15 % when transactions involve the majority of the partitions due to POCC's greater resource efficiency. POCC presents lower staleness than Cure* by two orders of magnitude.

3.2.1.11 Wren

Despite the existence of prior TCC designs, they either blocked reads, like Cure [4], or did not support sharding, like SwiftCloud [98]. In this regard, Spirovska, Didona, and Zwaenepoel presented Wren [84], the first TCC system to support both non-blocking reads and sharding.

Like in previous works, the authors assume a multi-versioned distributed key-value store split into N partitions and fully replicated across M DCs. Furthermore, they assume that any replica can update a key for which it is responsible (multi-master).

The main novelty of Wren resides in its transactional protocol, with which the snapshot made visible to the client is the union of a stable snapshot installed in every partition of the DC and a client cache. With this strategy, Wren avoids blocking reads, in opposition to previous designs that assign snapshots that may not have been installed yet in all the partitions of the DC, which is the case of GentleRain [25] and Cure [4]. Furthermore, by equipping each client with a private cache, Wren ensures that client writes not yet reflected in the stable snapshot are also visible to the client.

With Wren's stabilization protocol (Binary Stable Time), partitions within a DC periodically exchange the latest local and remote commit timestamps they have applied to determine which local and remote updates can be made visible, i.e., the Local Stable Time (LST) and the Remote Stable Time (RST). In contrast with systems that use a single timestamp to track dependencies, this protocol enables local items to become visible independently of the visibility of remote items.

Additionally, Wren leverages a new dependency tracking protocol (Binary Dependency Time) where each write's dependencies are depicted by two timestamps, representing dependencies on remote and local updates.

Similarly to Cure [4], Wren's programming interface does not provide a specific operation for simple reads or writes. Instead, clients must issue an operation to start a transaction, issue the desired read and write operations for a set of keys and commit the transaction.

A partition keeps a version vector (VV) that stores the timestamp of the last applied transactions on each replica, including itself. Furthermore, it stores lst and rst, computed periodically through the stabilization protocol. Partitions additionally store two queues required for the 2PC protocol, which is used to commit the transactions. One queue stores the prepared transactions, and another is used for the committed transactions. For timestamping, each server uses its HLC.

The client state comprises a private cache (WC) that stores the versions written by the client that may not yet be reflected in the stable snapshot. Additionally, a client maintains the commit time of its last update transaction (hwt), and the metadata and data of the current transaction (i.e., its id, local and remote stable snapshots (lst and rst), write set, and read set).

To start a transaction, the client sends the lst and rst of its last transaction to a coordinator partition, which updates its lst and rst and proposes a snapshot at least as fresh as the one accessed by the client in previous transactions. The server sends the transaction's id and timestamps back to the client, which uses lst to prune the cache. Wren enforces the rst assigned to a snapshot to be lower than its lst so that the client does not have to deal with conflicting updates and can directly read from his cache the versions that remain after pruning.

For each key to read, the client checks the transaction's write set, read set, and then the cache, ensuring read-your-writes and repeatable reads. If a read cannot be served locally, the client issues a request to the concerned partitions together with the transaction's snapshot. Each partition updates its *lst* and *rst* and returns to the client the versions with the highest timestamp enclosed in the transaction's snapshot.

Writes are stored in the client's write set and sent in the commit request along with the transaction's id and *hwt*. The commit protocol is based on 2PC. First, the coordinator contacts the partitions responsible for the updated keys. These partitions update their clocks and propose a commit timestamp, which must be higher than the transaction's timestamps and *hwt* to reflect causality. Then, they send the proposed commit time to the coordinator and add the transaction to the respective queue. In the second phase, the coordinator selects the commit timestamp by computing the maximum of the received timestamps, sends it to the partitions and the client, and removes the transaction from storage. Upon the reception of the commit message, a partition updates its clock and moves the transaction to the committed queue.

Committed transactions are periodically applied in commit timestamp order and replicated whenever their commit timestamp is lower than any pending transactions. More specifically, the partition creates a new version of the key, stores it in the key's version chain, updates its version vector, and replicates the update. In the absence of updates, heartbeat messages are used to ensure the progress of the RST protocol.

Compared with Cure [4], which may block reads due to clock skew, Wren achieves up to a 2.33x improvement in response time and 25% in throughput in strongly skewed workloads. When compared with H-Cure, a version of Cure that uses HLCs, it also achieves better latency and throughput because, even though this version does not block due to clock skew, it may still block if a transaction is assigned a snapshot that is not installed on the partition. Wren reduces latency by up to 3.6x and 1.6x compared to Cure and H-Cure, respectively. Additionally, it improves throughput by 1.33x and 1.23x when compared with the same systems. It is also highly scalable concerning the number of partitions, presenting almost the ideal throughput improvement. The performance gains come at the cost of a slight increase in update visibility.

3.2.1.12 Contrarian

Only two years since Lu *et al.* presented COPS-SNOW [56], the first system to implement latency-optimal ROTs, Didona *et al.* [23] proved that COPS-SNOW protocol induces an extra overhead on writes that jeopardizes performance even in read-dominated workloads. More precisely, their work shows that this overhead grows linearly with the number of clients due to the *readers check* procedure (described in section 3.2.1.8).

With this in mind, Didona *et al.* present the Contrarian protocol, which respects the non-blocking and single-version *SNOW properties* [56] but requires two rounds of communication. However, it has the advantage of not imposing any overhead on writes. Similarly to GentleRain

[25] and Cure [4], Contrarian relies on a stabilization protocol to track causality. However, in contrast with the systems mentioned earlier, it relies on dependency vectors instead of scalar timestamps.

Like previous designs, Contrarian targets a fully replicated key-value store. For each version of a key, a dependency vector stores the last update timestamp it has seen from each DC. Likewise, each client keeps a dependency vector with identical structure and size, which tracks the versions on which it depends. Upon a write operation, the client's dependency vector is appended to the request and copied to the corresponding remote entries of the dependency vector of the key version. The timestamp of the new version, which must be higher than the ones in the client's dependency vector to ensure causality, is then used to set the entry of the local DC. Replication is handled asynchronously to remote DCs.

Additionally, each DC independently calculates a vector of cutoff timestamps (referred to as *GSS*) that tracks the lower bound of the remote versions received in the DC. For that, partitions maintain a version vector with the last received update from each DC. These version vectors are periodically exchanged among the partitions of a DC to compute the *GSS*.

These procedures and data structures ensure that clients can only read stable versions (i.e., whose dependencies have already been received in the DC). This condition holds when remote entries in the dependency vector of the version are lower or equal to the respective entries in the *GSS* of the DC's partition responsible for the key.

Contrarian's ROTs take two rounds to complete. First, the client selects a coordinator from the partitions that store the keys he wishes to read and sends its dependency vector to that partition. The server then calculates the snapshot vector as follows: the local entry is set to the maximum between the coordinator's clock and the highest local timestamp seen by the client; the remote entries are set to the "entry-wise maximum between the *GSS* at the coordinator, and the dependency vector of the client" [23, p. 1622]. The snapshot is returned to the client, which forwards it to the appropriate partitions, along with the keys to read. For each key, the version returned to the client will be the one with the highest timestamp that belongs to the snapshot.

Unlike previous solutions [25, 4], which use physical clocks and thus need to block when the ROTs' timestamps are higher than the partition's clock, Contrarian uses HLCs to ensure the non-blocking property.

When evaluated against COPS-SNOW [56], Contrarian achieves better latency due to the higher replication cost that results from communicating the dependency lists and to COPS-SNOW's *readers check* procedure. Contrarian only shows higher latencies than COPS-SNOW in under-utilization scenarios.

3.2.1.13 PaRiS

Spirovska, Didona, and Zwaenepoel [85] presented PaRiS, the first TCC system built on a partially replicated data store that supports non-blocking ROTs.

Implementing TCC with non-blocking reads in a partially replicated data store is not trivial because servers of different DCs may concurrently serve different reads of the same transaction. Conversely, a transaction executes entirely within a single DC in a fully replicated scenario.

To address this, PaRiS relies on a novel causal dependency tracking protocol, UST. This protocol finds and retrieves a stable snapshot installed in all DCs, ensuring that ROTs do not block. PaRiS uses a client-side cache, in which the client stores his updates that the snapshot identified by the UST protocol does not yet reflect. This way, it ensures that clients observe monotonically increasing snapshots. To improve the freshness of the snapshot determined by UST over a solution that uses logical clocks, which can advance at very different rates on different partitions, PaRiS uses HLCs.

Like previous systems, PaRiS assumes a multi-versioned distributed key-value store split into N partitions and replicated in M DCs. However, as they consider a partially replicated data store, a DC may only comprise a subset of the dataset (R, the replication factor, is lower than M). Hence clients may need to contact remote replicas for specific keys. PaRiS also assumes that any replica can update a key for which it is responsible (multi-master system).

PaRiS's shares the same programming interface as Wren [84], providing operations to start a transaction, read and write a set of keys, and commit the transaction.

A partition keeps a version vector (VV) that stores the timestamp of the last applied transactions for each replica, including itself. Periodically, each partition exchanges the minimum of its version vector with other local partitions to compute the local DC's Global Stable Time (GST). The nodes within a DC are organized as a tree to reduce message overhead, and the GST is aggregated from the leaves to the root. The root of each DC exchanges the DC's GST to compute the minimum of all exchanged values (ust). ust identifies a timestamp such that all transactions with lower timestamps have been applied in all DCs. The roots share this value with all the other nodes of their DC. This protocol identifies a stable snapshot with a single timestamp. Partitions additionally store two queues required for the 2PC protocol, which is used to commit the transactions. One queue stores the prepared transactions, and another is used for the committed transactions.

PaRiS equips each client with a cache (WC_c) that stores the versions written by the client that may not yet be reflected in the stable snapshot. Additionally, a client maintains the highest stable snapshot timestamp he knows (ust_c), the commit time of its last update transaction (hwt_c), and the metadata and data of the current transaction (i.e., its id, write set (WS_c) and read set (RS_c)).

Whenever a client issues a transaction, it sends ust_c to the selected coordinator partition, which updates its utc and assigns a snapshot timestamp ts at least as fresh as the one accessed by the client in previous transactions. The server assigns an id to the transaction, stores it, and sends a response to the client, including the transaction's id and timestamp. The client uses the received timestamp to remove the versions with timestamp lower or equal to that value from his cache. Then, for each key to read, the client first checks the transaction's write set, followed by the read set and then its cache, ensuring read-your-writes and repeatable reads. If a read cannot be served locally, the client issues a request to the coordinator, which sends the request to the involved partitions together with the transaction's timestamp. The cohorts update their ust and return the version of

the keys with the highest timestamp below the snapshot timestamp. The coordinator sends them back to the client, who adds them to RS_c.

Writes are stored in WS_c and sent in the commit request along with the transaction's id and hwt_c. The commit protocol is based on 2PC. First, the coordinator contacts the partitions responsible for the updated keys. These partitions update their clocks to be at least as high as the maximum of the transaction's timestamp and hwt_c. Then, they update *ust*, send the proposed commit time to the coordinator (i.e., the maximum of his clock and *ust*), and add the transaction to the respective queue. In the second phase, the coordinator picks the commit timestamp, the maximum of the received timestamps, sends it to the partitions and the client, and removes the transaction from storage. Upon the reception of the commit message, a partition updates its clock and moves the transaction to the committed queue.

Committed transactions are periodically applied and replicated whenever their commit timestamp is lower than pending transactions. More specifically, the partition creates a new version of the key, stores it in the key's version chain, updates its version vector, and replicates the update. In the absence of updates, heartbeat messages are used to ensure the progress of the UST protocol.

PaRiS evaluation shows it achieves the same latency for both read-heavy and write-heavy workloads compared to an eventually consistent system. Additionally, it proves that PaRiS scales well with the number of DCs. Compared to a blocking system, the worst-case difference between the visibility latency is around 200ms. Overall, PaRiS achieves low latency, low storage requirements, and rich transactional semantics at the cost of slight data staleness.

3.2.1.14 Eiger-PORT

Guided by the *NOCS theorem* (described in section 2.2.2), which suggests the possibility of further optimizing systems with ROTs, Lu, Sen, and Lloyd [58] proposed a novel system design, PORT. This design features performance-optimal ROTs without sacrificing the write's performance, contrary to what happens in COPS-SNOW [56]. Furthermore, in systems with only simple writes, this design is compatible with process-ordered serializability, a consistency model that combines serializability and sequential consistency. On the other hand, in the presence of WOTs, namely when applied to Eiger [54], PORT's design provides CC while enabling significant performance improvements due to its performance-optimal ROT algorithm. More specifically, while Eiger's original algorithm takes up to three rounds of communication and uses linear-sized metadata, Eiger-PORT's ROTs terminate in a single round using constant metadata.

The core of PORT's design lies in version clocks. These logical clocks capture both the ordering constraints between requests and the stable frontier, the most recent snapshot in which all writes are in the stable region (i.e., "where all writes have committed and system states are finalized" [58, p. 336]). In particular, the version clock of a client tracks the minimum write versionstamp of all the servers he has contacted. For reads to be able to read stale values, the servers also comprise a multi-versioning framework, which stores written values indexable through versionstamps. The clock is advanced when a response to a write is received, ensuring that subsequent reads have a greater versionstamp than the write (read-your-writes).

In Eiger-PORT, clients keep a *lstmap*, which tracks the local safe time (*lst*) of each server, and a global stable time (*gst*) which is the minimum *lst* across all servers. Both of these values are Lamport timestamps. When the client issues a ROT, it uses *gst* as the read timestamp. When issuing a WOT, the *gst* is also attached to specify the stable frontier it causally depends on. The client updates *lstmap* after each read/write request.

The proposed WOT algorithm uses the 2PC protocol and is similar to the one used in the base system. Upon a WOT request, the coordinator records the current Lamport time, creates a new pending version, and saves it in *pending_wtxns*, which tracks ongoing WOTs in order of pending time. The minimum of the time in *pending_wtxns* is the *lst* of the server and indicates that there are no pending writes before that timestamp. In the prepare phase of the protocol, each cohort sends a *prepared_time* to the coordinator, which calculates the maximum of all received times to define the commit time and sends it to the cohorts. The cohorts receive the commit time and update their *lst*.

Upon a ROT, the coordinator finds the version at the requested versionstamp (the stable frontier) and verifies if there are any recent writes by the same client. If there are, then they must be returned to ensure read-your-writes, thus logically moving the client's writes before the stable frontier. In case the version at the requested versionstamp was written by the client, it checks if there exists any version between the version's snapshot time and the version's commit time. If there is, it is returned to ensure write isolation.

The evaluation of Eiger-PORT reflects a significant performance improvement of its transactional base system that results from the reduced number of messages and metadata. In particular, the comparison between these systems shows up to three times throughput improvement and 60% latency reduction, at the cost of some data staleness.

Furthermore, Eiger-PORT provides performance-optimal ROTs because it reads at the stable frontier from a pre-determined snapshot and only uses one versionstamp per read request.

3.2.1.15 Summary

This section reviewed several causally consistent distributed systems that support ROTs.

COPS [53] was the first causally consistent system to support sharding within each replica. It tracks the dependencies of each operation in metadata and verifies them upon a ROT.

Eiger [54] extends COPS with a richer column-family model and WOTs. It avoids the metadata explosion of COPS by keeping track of dependencies on operations either than on values, which also results in higher performance.

ChainReaction [5] uses a new variant of chain replication where writes may return as soon as the first k replicas process them, ensuring fault tolerance. To order the operations, it uses a sequencer process within each replica. On read-heavy workloads, ChainReaction exhibits higher throughput and requires fewer metadata than COPS.

Orbe's [24] key novelty is the usage and compression of dependency matrices, which result in increased throughput compared to COPS.

GentleRain [25] was the first system to employ a stabilization-based technique and to introduce the trade-off between visibility and throughput. The comparison between GentleRain and COPS shows that GentleRain yields a considerably higher throughput for read-heavy workloads, as COPS needs to track more dependencies.

SwiftCloud [98] uses a log-based approach where client-side replicas keep a partial set of the data store. Therefore, operations can be handled locally with minimal latency. To tolerate DC failures, SwiftCloud's clients only depend on updates that are stable in at least k DCs. SwiftCloud also supports CRDTs.

Cure [4] was the first system to provide TCC. It supports CRDTs and uses a stabilization strategy similar to GentleRain. However, it uses a coordinator-based approach where the client must first retrieve the stable snapshot of the transaction. By employing vector clocks instead of scalar timestamps to track the stable snapshot, Cure achieves less staleness at remote sites than GentleRain, introducing a new trade-off between staleness and metadata size. In terms of throughput, Cure outperforms Eiger as the update rate increases due to the overhead incurred by the dependency checks.

COPS-SNOW [56] extends COPS with latency-optimal ROTs by incurring extra costs on writes. Compared with COPS, it shows an improvement in latency for ROTs at the expense of lower system throughput.

Occult [65] adopts an optimistic approach where writes are immediately applied in the data store, trading off the availability of read operations for write visibility. Occult also presents a novel way to track causality using shardstamps and causal timestamps and emphasizes the trade-off between the causal timestamps' size and the reads' staleness.

Similarly to Occult, POCC [86] trades off availability for higher data freshness, always returning the most recent available version to the client, even if its dependencies have not yet been installed.

Wren [84] was the first system to provide TCC and support non-blocking reads and sharding — Cure could block reads due to clock skew, and ShiftCloud did not support sharding. Its main novelty resides in its transaction protocol, with which the snapshot made visible to the client is the union of a stable snapshot installed in every partition of the DC and a client cache. Compared with Cure, Wren reduces latency by up to 3.6x and improves throughput by 1.33x.

In a later work, Didona *et al.* [23] proved that COPS-SNOW's protocol induces an extra overhead on writes that jeopardizes performance even in read-dominated workloads. With this in mind, they proposed Contrarian [23], which respects the non-blocking and single-version *SNOW properties* [56] but requires two rounds of communication. The system relies on a stabilization protocol to track causality. However, in contrast with Cure and GentleRain, it uses a HLC to timestamp the operations. Contrarian achieves higher latency than COPS-SNOW in most scenarios.

PaRiS [85], was the first TCC system built on a partially replicated data store that supports non-blocking ROTs by relying on a novel causal dependency tracking protocol, UST, which retrieves a stable snapshot installed in all DCs. Furthermore, like Wren, it uses a client-side cache to ensure

read-your-writes and a HLC to increase the freshness of the snapshots. PaRiS achieves similar latencies to an eventually consistent system at the cost of slight data staleness.

Eiger-PORT [58] improves Eiger's original algorithm, which takes up to three rounds of communication and uses linear-sized metadata, by terminating in a single round and using constant metadata. Its design features performance-optimal ROTs without sacrificing the write's performance, contrary to what happens in COPS-SNOW. The core of PORT's design lies in version clocks, a new logical clock that captures both the ordering constraints between requests and the stable frontier. It improves Eiger's throughput up to three times and reduces latency up to 60%, at the cost of increased data staleness.

Overall, most systems assume a similar architecture: a fully replicated key-value store split into N partitions, each replicated at all M DCs. SwiftCloud supports client-side partial replication, but only PaRiS supports partial replication on server-side replicas. Furthermore, multi-versioning has proven essential for concurrency, isolation, and read performance in these systems because it allows returning stale versions to the client without waiting indefinitely. However, it is insufficient to ensure performance-optimal ROTs: computing a stable snapshot may require extra off-path messages and metadata, and blocking reads may be necessary due to clock skew. Finally, we also observed a trade-off between staleness and several other properties, namely availability, performance, and the size of the metadata that encodes a version's dependencies.

The following section classifies these systems through the lens of the *NOC properties*.

3.2.2 Classification of Existing Causally Consistent Systems

The *NOC properties* described in section 2.2.2 provide a way to analyze existing systems and identify room for improvement in their design. In this regard, table 3.3 categorizes the systems reviewed in the previous section according to these properties. It also identifies the clock used, the operations provided, and the type of replication supported.

As shown in table 3.3, some systems, such as GentleRain [25], Cure [4], ChainReaction [5], Orbe [24], and POCC [86], block ROTs until a consistent view of the data is available, thus penalizing availability and latency. GentleRain [25], for example, blocks ROTs until the global stable time (i.e., the last update timestamp seen by all the partitions at the DC) advances past the dependency time of the client (i.e., the last update timestamp seen by the client). This way, it can guarantee that the snapshot reflects any values previously seen by the client that requested the operation.

Except for SwiftCloud [98] and PORT [58], all the systems presented in table 3.3 require more than one round of messages to perform a ROT. Even though most systems' ROTs take a bounded number of rounds, in ChainReaction [5] and Occult [65], a transaction can be aborted, resulting in an unbounded number of on-path messages, which has a similar effect to blocking. Other algorithms, such as the ones used in COPS-SNOW [56], GentleRain [25], and Cure [4], use additional off-path messages "whose removal affects only the correctness of read-only transactions" [58, p. 335]. COPS-GT [53] and Eiger [54], on the other hand, use more than one on-path round. Furthermore, many systems, particularly Orbe [24], Cure [4], Wren [84], Contrarian [23],

PaRiS [85], and POCC [86], adopt a coordinator-based approach, where "a coordinator partition is responsible for assigning a stable snapshot to a transaction that starts" [85, p. 307]. This strategy results in two rounds of communication, the first to get the stable snapshot and the second to request the values of the keys to read. On the other hand, Eiger-PORT [58] always completes a ROT in a single round as clients specify the version from where they want to read. However, in the "worst case scenario where every read-only transaction is issued by a brand new (or inactive) user" [57, p. 24] and "there is no bound on clock-skew"[57, p. 24], Eiger-PORT's approach would, on a first request, return an arbitrarily stale value of the data, whereas in other systems the coordinator establishes the stable frontier from which the read can be performed. In SwiftCloud [98], reads are performed from the base version of the data store that the client holds, so they always succeed locally without any synchronous communication with the DC provided that the client requests keys from his interest set. Asynchronously, the client's local state is updated through a notification session established with a DC, and the client's updates are transmitted to a DC through a log-based approach.

The table also shows that most algorithms require metadata to be exchanged upon a ROT to compute a consistent view. In COPS [53], Eiger [54], and COPS-SNOW [56], the metadata size required in a ROT depends on the number of dependencies of the requested keys. In COPS-GT [53], for example, in the first round of communication, the client issues a request for each key in parallel to his local cluster, which returns a $\langle value, version, deps \rangle$ tuple where $deps$ is the list of dependencies of the key. On the other hand, in Cure [4], POCC [86], and Contrarian [23], the metadata is linear in size concerning the number of DCs because the client exchanges his dependency vector with the coordinator, and this vector has one entry per DC. GentleRain [25], Orbe [24], Wren [84], PaRiS [85], and Eiger-PORT [58] use constant metadata, particularly one or two timestamps. Swift Cloud's [98] ROTs are performed locally and, thus, do not synchronously exchange any metadata.

Smaller metadata, however, results in false dependencies, which inevitably leads to higher data staleness. Furthermore, as Kakwani and Nasre argue: "even though the metadata-size required for ROT in some systems is smaller, that cost is transferred to periodic metadata exchange messages" [40, p. 5]. In Wren [84], for example, partitions within a DC exchange the commit timestamp of the latest local and remote transactions they have applied. Similarly, in PaRiS [85], partitions periodically exchange the minimum of their version vectors to compute the global stable time of the local DC [85]. On the other hand, in SwiftCloud [98], updates are asynchronously transferred from the clients to a DC, and a notification protocol is used to send updates to the clients according to their interest set.

All systems that use physical clocks [5, 24, 25, 4, 86] are blocking because the clock "cannot be moved forward to match the timestamp of an incoming ROT" [23, p. 1622]. To overcome this, some protocols rely on HLCs [85, 84, 23] while others use logical time [53, 54, 65, 56, 58], namely timestamps, vector clocks, version vectors, or variants of these mechanisms.

Overall, most of the systems provide ROTs. Only Eiger [54] and Eiger-PORT [58] provide WOTs, whereas SwiftCloud [98], Cure [4], Occult [65], Wren [84], and PaRiS [85] support

System	Taxonomy	Clock	Replication	ROT performance optimality		
				Non-blocking	Rounds	Metadata
COPS [53]	ROT	Logical	Full	✓	≤ 2	$O(D)$
Eiger [54]	ROT & WOT	Logical	Full	✓	≤ 3	$O(D)$
ChainReaction [5]	ROT	Logical & Physical	Full	✗	≥ 2	$O(M)$
Orbe [24]	ROT	Logical & Physical	Full	✗	2	$O(1)$
GentleRain [25]	Snapshot & ROT	Physical	Full	✗	$\leq 2 + off\text{-}path$	$O(1)$
SwiftCloud [98]	Generic	Logical	Partial*	✓	0	$O(1)$
Cure [4]	Generic	Physical	Full	✗	$2 + off\text{-}path$	$O(M)$
Occult [65]	Generic	Logical	Full	✓	≥ 1	$O(N)$
COPS-SNOW [56]	ROT	Logical	Full	✓	$1 + off\text{-}path$	$O(D)$
POCC [86]	ROT	Physical	Full	✗	2	$O(M)$
Wren [84]	Generic	Hybrid	Full	✓	2	$O(1)$
Contrarian [23]	ROT	Hybrid	Full	✓	2	$O(M)$
PaRiS [85]	Generic	Hybrid	Partial	✓	2	$O(1)$
Eiger-PORT [58]	ROT & WOT	Logical	Full	✓	1	$O(1)$

Table 3.3: Characterization of geo-replicated causally consistent systems that support ROTs. Partial* stands for partial replication at the client. D stands for the number of dependencies, M is the number of DCs and N is the number of partitions. (Adapted from [58], [85] and [23]).

generic read/write transactions. GentleRain [25] provides a snapshot that may not reflect the last reads of the client.

While most systems only support full replication, PaRiS [85] is the only system that supports partial replication on the server side. SwiftCloud [98], on the other hand, allows client replicas to declare the items they want to receive updates from but requires server-side replicas to store a full copy of the data.

Finally, considering the *NOC properties*, only Eiger-PORT [58] and SwiftCloud [58] exhibit the performance-optimal properties of ROTs. Despite this, SwiftCloud [98] does not support sharding, and PORT's [58] data freshness relies on the assumption that clients can determine the timestamp from where to read, either because they have performed previous transactions on the same client session or by leveraging information from other client's transactions conducted on the same machine [57].

3.2.3 Recurrent Strategies to Implement Causally Consistent Systems

One of the main challenges in causally consistent distributed systems is ensuring causality in scenarios where data is spread across partitions and replicated across DCs. In order to avoid the overhead of synchronous replication on each request, most systems opt for asynchronous replication. However, this decision exacerbates the challenge of guaranteeing that operations are applied in causal order within each DC, as it becomes possible that a new version of a data item may arrive at a remote DC before its causal dependencies.

In this regard, the literature outlines several strategies to implement causally consistent systems, which generally fall into one of the following four: dependency checking, sequencer-based, stabilization protocols, and optimistic approaches. In the following subsections, we outline these

strategies and characterize causally consistent systems based on the current state of the art. Table 3.4 summarizes each strategy and its limitations.

Strategy	**Summary**	**Limitations**	**Examples**
Dependency Checking	Encode causal dependencies in metadata, store them together with the corresponding version or operation and send them upon replication of writes. Dependency check messages are used to ensure that updates are only installed before their dependencies.	Metadata and dependency checks cause communication overhead.	COPS [53], Eiger [54], COPS-SNOW [56], Orbe [24].
Sequencer-based	Operations are totally ordered in each replica.	Blocking; Scalability constraints.	ChainReaction [5], SwiftCloud [98]
Stabilization Protocols	"In general, each DC establishes a cutoff timestamp below which it has received all remote versions" [23, p. 1621]. Only versions with a timestamp lower than this cutoff timestamp can be made visible. Other variants use a vector with one timestamp per DC.	Normally combined with a coordinator-based approach, requiring at least two rounds of communication in ROTs; Blocking (when using physical clocks), but non-blocking otherwise.	GentleRain [25], PaRiS [85], Cure [4], Contrarian [23], Wren [84].
Optimistic	Are based on the optimistic assumption that the dependencies of a data item will have been received in a remote DC by the time the client wants to access them. The client is responsible for enforcing causality.	Blocking or require retry; Unbounded number of rounds; Reduces data staleness; Reduces coordination overhead in the servers.	POCC [86], Occult [65].

Table 3.4: Recurrent strategies to enforce causality.

3.2.3.1 Dependency Checking

A possible strategy to enforce causality is to encode causal dependencies in metadata and verify them before applying an operation (a technique often named dependency checking). In general, with this strategy, causal dependencies are sent along upon replication of a write operation. The receiving DCs verify if those dependencies are installed by sending dependency check messages to the responsible partitions. The new version can only be installed when the DC has installed all the dependencies.

Examples of systems that rely on dependency checking are COPS [53], COPS-GT [53], Eiger [54], Orbe [24], and COPS-SNOW [56]. COPS-GT [53], for example, associates each version of a key with a list of dependencies (i.e., other keys that causally precede it and their respective version). In contrast, Eiger [54] tracks dependencies on operations. Moreover, while COPS [53], Eiger [54], and Orbe [24] check dependencies upon a ROT to ensure a unified view of the data, COPS-SNOW [56] shifts the dependency checking to write operations. This way, COPS-SNOW

[56] makes the ROT algorithm latency-optimal. Furthermore, while COPS-GT [53] tracks and stores explicit dependencies, Eiger [54] and Orbe [24] use logical time.

Several systems that employ this technique incorporate optimizations to reduce the number of dependencies and dependency check messages to reduce storage and communication overhead, hence minimizing performance degradation. COPS [53] and Eiger [54] only track the nearest dependencies and one-hop dependencies, respectively. COPS-GT [53], on the other hand, requires all the dependencies to support ROTs and thus relies on a different solution, garbage collection, to discard old dependencies when they are no longer necessary for correctness. Another example is Orbe [24], which uses a sparse dependency matrix to store the nearest dependencies. Moreover, by leveraging the transitivity rule of causality, Orbe [24] resets the dependency matrix of a client session after each update operation.

Overall, despite various optimizations, the number of dependency check messages in the worst-case scenario "remains linear in the number of partitions" [25, p. 12]. However, compared to previous works [11, 45, 12, 75] that rely on log serialization and exchange, systems employing dependency checking do not require a single serialization point. Therefore, they allow partitioned data stores and eliminate the scalability constraints of previous works.

3.2.3.2 Sequencer-based

First solutions based on log serialization and exchange [11, 45, 12, 75] relied on a single serialization point in each replica to order the operations and then exchanged these logs "to establish potential causality and detect concurrency between operations at different replicas" [53, p. 404]. Therefore, they could only provide CC within single node replicas, i.e., they did not support partitioned data stores.

A later work [5] proposed a different sequencer-based solution that uses a central process in each replica. In particular, ChainReaction [5] uses a sequencer that orders the reads of a ROT and all writes to ensure a consistent view of multiple data items despite concurrent writes. This solution, however, affects scalability and performance, "increasing the latency of all update operations by one round-trip network latency within the data center" [24, p. 13]. Introducing an extra round of communication also inhibits the system from achieving performance optimality by affecting the *one-round* property.

SwiftCloud [98], on the other hand, adopts a log-based strategy where clients register updates locally in their log and asynchronously transfer them to a DC. DCs broadcast new updates and merge them into their logs. The system prunes the log through checkpointing to bind storage and network overhead.

Sequencer-based techniques impose a total order of operations on each replica, which may hinder horizontal scalability by inhibiting sharding [98] or centralizing the ordering in a single process [5].

3.2.3.3 Stabilization Protocols

Another common approach among the studied causally consistent systems is to employ a stabilization protocol [24, 25, 4, 85, 84, 23], which guarantees that the snapshot returned to the user has been applied across all DCs. To that end, in several timestamp-based algorithms, such as GentleRain [25], "each DC establishes a cutoff timestamp below which it has received all remote versions" [23, p. 1621]. Only versions with a timestamp lower than this cutoff can be made visible. In other systems, like Contrarian [23], dependency vectors, with one entry per DC, are used instead of scalar timestamps to track causality, and "the stabilization protocol determines, in each DC, a vector of cutoff timestamps" [23, p. 1621].

One of the challenges when using stabilization protocols is to ensure the non-blocking property of ROTs, which according to the SNOW [56] and NOCS [58] theorems (described in section 2.2.1 and section 2.2.2), is crucial to achieving latency-optimal or even performance-optimal ROTs. In particular, when using physical clocks, ROTs may be blocked due to clock skew to enable the clock to catch up with the transaction's snapshot timestamp and to ensure local and remote updates with a timestamp lower or equal to the transaction's snapshot timestamp have been installed.

To avoid blocking due to clock skew, Contrarian [23], Wren [84], and PaRiS [85] use HLCs, which allow partitions to advance their clock forward to match the timestamp of an incoming ROT, while still ensuring progress in the absence of events.

Furthermore, most stabilization-based systems use a coordinator-based approach, requiring an extra communication round for ROTs to request the stable snapshot to a coordinator. Hence, they disrespect the *one-round* NOC [58] property (described in section 2.2.2).

3.2.3.4 Optimistic

A smaller set of systems have adopted an optimistic approach to CC. Based on the optimistic assumption that the dependencies of a data item will have been received in a remote DC by the time the client wants to access them, "a server always returns the most recent available version of an item, even if some of its causal dependencies have not been replicated locally" [86, p. 527]. If, however, it is the case that the client requests a dependency of a data item that is missing, then the request must either be blocked by the server until the dependency becomes available, which happens in POCC [86], making the system vulnerable to network partitions, or retried by the client, like in Occult [65].

In other strategies, such as stabilization protocols and dependency checking, the response to a read includes the most recent version of the item whose causal dependencies are known to have been already replicated. These strategies increase the staleness of the data returned to clients and become susceptible to slowdowns when one component of the system is lagging behind the other. With this in mind, the optimistic strategy never delays writes to enforce consistency. Instead, it "shifts the responsibility for the enforcement of CC from the data store to its clients" [65, p. 453], which results in increased data visibility and "relieves the data store from the responsibility of tracking the delivery of updates" [86, p. 527-528].

3.2.3.5 Summary

This section outlined four main strategies for implementing causally consistent distributed systems, discussed their impact on the *NOC properties*, and classified the systems described in the previous section according to these strategies.

Section 3.2.3.1 described dependency checking, a strategy used by COPS [53], Eiger [54], Orbe [24] and COPS-SNOW [56] where causal dependencies are encoded in metadata and verified before applying an operation. Despite optimizations, the number and size of dependency check messages results in higher communication overhead.

Section 3.2.3.2 introduced the sequencer-based technique, which comprises log-based solutions, such as SwiftCloud [98], but also the strategy used by ChainReaction [5], where a single process totally orders the operations. This strategy may hinder horizontal scalability by imposing a total order of operations on each replica.

Section 3.2.3.3 addressed stabilization, the prevailing technique among the systems analyzed. This technique ensures the client performs a read from a causally consistent snapshot. To that end, it establishes a cutoff timestamp that identifies which updates can be safely made visible. When using physical clocks, this means that a ROTs may be blocked due to clock skew [25, 4], which leads to disrespecting the *non-blocking NOC property*. To avoid blocking due to clock skew, Contrarian [23], Wren [84], and PaRiS [85] use HLCs. However, given they adopt a coordinator-based approach, they still require an extra communication round, disrespecting the *one-round NOC property*.

Finally, section 3.2.3.4 described the optimistic approaches employed by Occult [65] and POCC [86]. In these strategies, updates are immediately made visible, attaining greater data visibility than previous strategies but potentially blocking or retying ROTs to ensure consistency with previously read versions.

3.3 Architectural Approaches to Causal Consistency

The previous sections described the algorithms and strategies used in existing causally consistent systems and classified them according to the *NOC properties* of ROTs. This section extends the review conducted thus far by describing two architectural approaches to CC. In particular, it describes Bolt-On's [8] layered approach, which upgrades existing cloud storage services with CC, proposing a storage-agnostic architecture that decouples the algorithmic implementation of CC from replication, liveness, and durability. Moreover, it analyzes the design choices that drove the implementation of CC in MongoDB [91], one of the first commercial databases to provide CC.

3.3.1 Bolt-on Causal Consistency

In the "Bolt-on Causal Consistency" paper, Bailis *et al.* [8] describe a layered and storage-agnostic approach to CC, presenting progress toward a general architecture for causally consistent systems. Their approach relies on a shim layer that provides CC atop a wide range of eventually consistent

storage systems. This "bolt-on" approach separates the durability, liveness, and replication concerns, already supported by existing storage services, from the consistency guarantees provided by the shim layer.

In Bolt-on's architecture, clients forward `get` and `put` requests to the shim layer, which works as a client-side library. For dependency tracking, their approach considers explicit causality, where the application must track the writes' causal history and provide it upon a `put` operation. Shim layers keep a local copy of the items in their local store, which is causally consistent by design, and use the interface provided by the Eventually Consistent Data Store (ECDS) to persist new data versions and propagate them to other shims.

Due to using an ECDS for write propagation, which generally has single-value register semantics, their approach must consider the problem of overwritten histories. For example, if $x_1 \rightarrow y_1$ and y_2 happen simultaneously, y_2 may overwrite y_1. If the shim only sees y_2, it may never learn about x_1. On the other hand, if $x_1 \rightarrow y_1 \rightarrow z_1$ and y_2 occur concurrently, and only the nearest dependencies are stored for each version, the shim may see z_1 and y_2 but not x_1, violating CC.

Considering this problem, the authors reformulate the correctness criteria for CC through causal cuts, which they define as follows: "The dependencies for each write in the causal cut should either i.) be in the cut, ii.) happen-before a write to the same key that is already in the cut, or iii.) be concurrent with a write to the same key that is already in the cut." [8, p. 765]. A new write may only be applied to the local store if it establishes a causal cut when merged with it.

Based on this novel definition of causality, the authors propose two algorithms. In the optimistic algorithm, writes are immediately applied in the local store and ECDS, and reads are satisfied locally. Asynchronously, a resolver process fetches the ECDS for newer versions of the keys that were recently read. In order to ensure the local store forms a causal cut, the resolver process must recursively fetch any missing dependencies of a new version before applying it.

The optimistic approach ensures fast reads and writes but may result in poor visibility. On the other hand, in the pessimistic algorithm, reads synchronously request the latest value from the ECDS, which may result in multiple rounds to get missing dependencies and penalize read latency.

Both approaches may result in metadata overhead because each write must be stored with its dependency set and keep metadata of the writing shims, represented by a vector clock.

When realized and evaluated using Cassandra as the underlying ECDS, the results suggest higher performance for the optimistic strategy, as reads are locally answered. For the pessimist approach, throughput and latency fall within 20% and 50% of the eventually consistent baseline, even though it attains better visibility than the optimistic approach. These results highlight the trade-off between staleness and read latency.

3.3.2 MongoDB

In 2019, Tyulenev *et al.* [91] discussed the architectural design of MongoDB, presenting one of the few industry implementations of CC to date. This work contributes with an applied solution of

CC that proves its applicability in large-scale industry databases. Furthermore, it combines state-of-the-art knowledge with new ideas to satisfy both the scalability and performance requirements and provide security and backward compatibility.

MongoDB supports replication and sharding. Each shard comprises one primary node, elected by consensus, and several secondary nodes. Primary nodes may accept both write and read operations, while secondaries are read-only nodes, which only receive updates from the primary node. Primary nodes handle writes by adding them to the operation log and applying them to their data set, adopting a state machine behavior. The log is replicated to the secondary nodes, which follow a similar behavior, applying the updates to their data sets.

When issuing read and write requests, clients can specify a concern. In the case of write operations, the concern specifies when a write can be acknowledged, thus enabling the client to manage the fault tolerance of its writes. In the case of reads, it allows clients to manage the consistency and isolation properties of the data read.

The primary node uses scalar timestamps generated by a HLC to version new writes and establish the log's order. Upon a write, the server returns the operation time to the client, which must attach it to subsequent requests. The node that receives the request must wait until its operation log catches up with the requested time. For security against malicious attacks, MongoDB's nodes sign the time returned to clients, avoiding the possibility of clients advancing the time inadvertently.

To ensure progress in the absence of writes, when a primary receives a request with a time not contained in its operation log, it uses a no-operation to advance the clock.

The evaluation of the system suggests that the impact of enabling CC when using a majority write concern has minimal impact on throughput. Additionally, it indicates that the read capacity increases with the number of available nodes.

3.3.3 Summary

This section reviewed two causally consistent architectures, which enable applications to manage some of the system's trade-offs and provide a general solution for implementing a causally consistent distributed system.

Bailis *et al.*'s [8] work presents progress toward standardizing a solution for implementing causally consistent distributed systems. They describe a storage-agnosic system architecture where a shim layer upgrades the consistency guarantees of the underlying storage system. Additionally, their work describes two alternative algorithms for guaranteeing CC, enabling the application to manage the trade-off between staleness and read latency.

Tyulenev *et al.* [91] described the design choices that drove the implementation of CC in MongoDB, proving CC's applicability in large-scale industry databases. MongoDB's design relies on HLCs to timestamp events and enables clients to manage the safety properties of read operations and the fault-tolerance of write operations by specifying additional requirements on operations.

3.4 Evaluation Metrics

Previous architectures for geo-replicated causally consistent systems were generally evaluated with microbenchmarks, which "constitute the first line of performance testing" [76]. "Microbenchmarks are used to measure simple and well-defined quantities such as elapsed time, rate of operations, bandwidth, or latency" [76].

Table 3.5 describes the metrics used to evaluate the implementations reviewed in section 3.2. Overall, these metrics do not vary much across the literature, and neither do the procedures used to assess them.

The two metrics that are typically assessed are the latency and throughput of each operation. For comparison, *ping* requests are sometimes used to establish the limits imposed by the hardware. In Orbe [24] and GentleRain [25], for example, the *Echo* operation is used for this purpose. Additionally, other implementations, with other consistency guarantees or strategies, are normally used for comparison. In some systems, such as COPS [53] and Eiger [54], the authors also measure tail latency using high percentiles, such as 95th, 99th, and 99.9th, which can help identify a small set of requests that are taking longer than the average. "Percentiles are often used in service level objectives (SLOs) and service level agreements (SLAs), contracts that define the expected performance and availability of a service" [43, p. 15]. A common approach is to assess these metrics in a saturated system.

In most studies, the system's throughput is characterized as a function of different variables. Commonly, one is the ratio of reads and writes, thus assessing how the system performs under read-heavy and write-heavy workloads. The throughput is sometimes assessed as a function of other workload parameters, such as the number of keys per read or write operation, the inter-operation delay, among others.

In order to quantify the scalability of the system, the throughput is typically measured for different numbers of DCs and partitions per DC.

Most systems also measure resource overhead, namely the amount of metadata exchanged, CPU usage, and memory usage. In PaRiS [85], for instance, the authors measure the amount of data exchanged within each DC and between DCs.

Finally, in PaRiS [85] and GentleRain [25], the staleness of the data is also measured by calculating the visibility latency of each update, i.e., the difference between the wall-clock time when an update becomes visible in a DC and the wall-clock time when it was received in that same DC.

3.5 Summary

Section 3.1 described the systematic review used to survey the state of the art in this dissertation's domain. It first outlined the methodology and the selected data sources. Then, it described the research questions (RQs) that drove the search process. **RQ1** focuses on identifying the ideal properties of read-heavy systems, namely the performance-optimal properties of ROTs. **RQ2**

Metric	Definition
Latency	Even though it represents the time that a request is waiting to be handled [43], in the context of distributed systems, it is often used to refer to the response time of the operation.
Tail Latency	Latency experienced by a small number of requests in a system. Often measured using high percentiles, such as the 95th, 99th, or 99.9th percentile, to identify the performance of a small number of requests that are experiencing delays significantly longer than the average request.
Throughput	The amount of operations that can be handled by the system in a given amount of time.
Scalability	"A measure of how adding resources (e.g. usually hardware) affects performance. A scalable system is one that allows you to add hardware and get a commensurate performance improvement". [29]
Message size	Amount of metadata exchanged between system components.
Number of messages	Number of messages sent between nodes in a given period of time.
Memory usage	Measures the amount of memory used by the system or component as a percentage of the total available memory.
CPU usage	Measures the amount of CPU time used by the system as a percentage of the total available CPU time.
Data Staleness	In PaRiS [85], this metric measures the time interval between the instant at which an update is received in DC_i and the instant at which the update becomes visible in DC_i. In GentleRain [25], it is the time interval between the instant at which the update is installed at its local DC and the time when it becomes visible at a remote DC.

Table 3.5: Metrics used to evaluate distributed systems.

seeks to identify the properties and strategies of the existing causally consistent systems that support ROTs. **RQ3** aims to identify the metrics typically used to validate and evaluate these systems. Then it presented the inclusion and exclusion criteria, the search query used in each engine, and the primary stages of the systematic review.

Section 3.2, which focused on **RQ1** and **RQ2**, described concrete implementations of several causally consistent systems with support for ROTs (Section 3.2.1), examined their properties (section 3.2.2), and explained the main strategies used to enforce causality (Section 3.2.3).

Section 3.2.1 described the causally consistent systems identified in the literature review. Most of them share a similar architecture, featuring a fully replicated key-value store split into N partitions, each replicated at all M DCs. In contrast, SwiftCloud [98] supports client-side partial replication, and PaRiS [85] supports partial replication on the server side. Furthermore, multi-versioning stood out as a critical factor for concurrency, isolation, and read performance. However, multi-versioning alone fails to ensure performance-optimal ROTs because computing a stable snapshot may require extra off-path messages and metadata, and clock skew may lead to blocking. Additionally, a trade-off was observed between staleness and several other properties, namely

availability, performance, and the size of the metadata that encodes a version's dependencies.

Section 3.2.2 examined existing causally consistent systems through the lens of the performance-optimal properties of ROTs. It addressed how the properties of the ROT algorithms of each system relate to the type of clock they use, analyzed their interdependencies, and pointed out the leading causes for violating each property. The usage of physical clocks is the main reason for blocking ROTs. Except for Eiger-PORT [58], which provides single-round ROTs, and SwiftCloud [98], where the client-replica answers the read requests, all algorithms take more than one round to complete. However, only those where a transaction may be aborted present an unbounded number of rounds. Some systems use extra off-path communication to coordinate a consistent view across shards, and several systems adopt a coordinator-based approach, which results in two rounds of communication. In most algorithms, the metadata used in ROTs grows linearly with the number of DCs due to the exchange of vector clocks or version vectors. ROT algorithms that use constant metadata usually require extra rounds of communication. PaRiS [85] and Wren [84] deliver two out of three of the performance-optimal properties of ROTs. Their ROT algorithms are non-blocking and use constant metadata, but they both use two rounds of communication. Additionally, PaRiS [85] provides partial replication, while Wren [84] assumes a fully replicated data store. Finally, considering the *NOC properties*, only Eiger-PORT [58] and SwiftCloud [58] exhibit the performance-optimal properties of ROTs. SwiftCloud [98], however, does not support sharding.

Section 3.2.3 described the main strategies used to ensure CC in replicated and partitioned systems. Dependency checking, which consists in encoding causal dependencies in metadata and verifying them before applying an operation, leads to an extra communication overhead due to the number and size of dependency check messages. Sequencer-based techniques impose a total order of operations on each replica, which may hinder horizontal scalability. Stabilization protocols are another strategy where, in general, a cutoff timestamp is defined within each DC, and only updates below this timestamp are made visible. It usually requires extra rounds of communication to define the cutoff timestamp. Finally, in systems that adopt an optimistic approach, clients are the ones to enforce causality, and updates are immediately applied, which reduces data staleness but may lead to blocking if missing dependencies are requested. Eiger-PORT [58] does not employ any of these strategies because each trades off at least one of the desired properties of ROTs.

Then, section 3.3 reviewed two causally consistent architectures, which enable applications to manage some of the system's trade-offs and present progress toward standardizing an architecture for causally consistent distributed systems. Bailis *et al.*'s [8] work describes a layered and storage-agnostic approach to CC, where a shim layer upgrades the consistency guarantees of an ECDS. Tyulenev *et al.* [91] described the design choices that drove the implementation of CC in MongoDB, which enables clients to manage the safety properties of read operations as well as the fault-tolerance of write operations by specifying read and write concerns.

Section 3.4, which focused on **RQ3**, identified the metrics used in the reviewed systems. All the analyzed systems use microbenchmarks to measure the latency of the operations and evaluate the system's throughput as a function of different variables (e.g., the read/write ratio). Further-

more, scalability is usually measured by varying the number of partitions and DCs. Data visibility, which is often traded off for higher latency and throughput, is also commonly assessed by calculating the visibility latency of each update.

Chapter 4

Problem Statement

Contents

The previous chapter reviewed the state of the art of causally consistent distributed systems, analyzing their properties and strategies and identifying the metrics used to evaluate them. This chapter formalizes the problem under investigation. First, section 4.1 explains the open problems, and section 4.2 defines the scope of this dissertation. Section 4.3 presents the hypothesis this dissertation seeks to validate. Section 4.4 presents the research questions that will drive this research. Section 4.5 describes the methodology used to validate the hypothesis. Finally, section 4.6 summarizes the topics covered in this chapter.

4.1 Open Problems

CC has been an attractive consistency model for achieving high availability, performance, and intuitive behavior in geo-replicated systems because it does not incur the overhead of stronger consistency models or the ordering anomalies of EC. In this regard, the research community has proposed several causally consistent system implementations and fewer architectural designs, such as the ones surveyed in this work. More recently, the prevalence of reads in real-world workloads [15, 18, 69, 92, 100] has encouraged some of these works to focus on improving read performance within causally consistent systems. However, fewer studies have approached the applicability of these findings to the industry.

Existing research focuses on the algorithmic design to ensure causality, primarily within key-value data stores, which serve as a fundamental building block for many systems but make building

several services arduous due to their basic construction [54]. In Eiger [54], the authors addressed part of this problem by providing a richer data model. However, another core issue resides in the lack of general solutions for applying CC to real-world read-heavy systems, pointing toward the need to extrapolate the literature's knowledge into a reference architecture that generalizes to the broad class of read-heavy systems. Ideally, to leverage the added benefits of existing cloud storage services, the literature's findings should be incorporated in a storage-agnostic way, akin to Bailis *et al.* Bolt-on approach.

Additionally, current causally consistent systems have been evaluated for demonstration purposes (e.g., using microbenchmarks). However, there remains scarce evidence on how they perform when realized in real-world systems that use data to enforce business rules rather than for pure data management.

Concerning the characteristics of existing causally consistent systems, the *NOC properties* of ROTs provide a new baseline to analyze previous systems and identify room for improvement in their design. In particular, the analysis performed in section 3.2.2 indicates that only Eiger-PORT [58] supports performance-optimal ROTs in a sharded system. Hence, all other causally consistent systems are potential candidates for improvement. Besides, as Eiger-PORT [58] assumes full replication of the data, to the best of our knowledge, no system provides performance-optimal ROTs and server-side partial replication, indicating the need to assess the possibility of achieving both properties.

Finally, even though multi-versioning is inherent to several causally consistent systems, making it possible to achieve greater concurrency, isolation, and support stale reads, it can be further exploited to improve the system's auditability by integrating and expanding the core ideas of replicated state machines (like Viewstamped Replication [71, 51]) with stronger *value semantics*. By integrating these ideas, the system can be perceived as a succession of deterministic, atomic states, enabling developers to reason about the system at a stable point in time.

4.2 Scope

In light of the shortcomings outlined in section 4.1, the present dissertation aims to assess whether the results achieved in existing causally consistent key-value stores can be extrapolated into a reference architecture that generalizes to the broad class of read-heavy systems. It seeks to identify the building blocks and strategies that enable optimal ROT performance in geo-replicated read-heavy systems and outline how they must be assembled regardless of the service provided. In addition, it strives to incorporate the literature's findings in a storage-agnostic way that can leverage the benefits of existing cloud storage services, namely their high availability, data accessibility, durability, and reliability. Lastly, it aspires to improve the auditability of these systems by leveraging *value semantics*.

4.3 Hypothesis

In order to address the problems identified in section 4.2 and to achieve the goals outlined in section 1.4, the present dissertation aims to validate the following hypothesis:

> *There exists a reference architecture that (1) manifests the ideal properties of geo-replicated causally consistent read-heavy systems, (2) upgrades the consistency guarantees of existing cloud storage services, and (3) enables value semantics, thereby facilitating auditing and enabling developers to reason about the system's state and data at a point in time.*

This work aims to provide a solution for delivering CC in read-dominant workloads. To that end, it proposes to provide a storage-agnostic reference architecture for causally consistent read-heavy systems. This solution can empower enterprises with the necessary tools to expedite their systems' development and leverage the availability, performance, and intuitive behavior of CC.

Moreover, this dissertation aims to prove that the reference architecture can be designed to:

1) **Manifest the ideal properties of geo-replicated causally consistent read-heavy systems.**
   In order to meet customer expectations and service requirements and to increase user engagement, the realized reference architecture must deliver the best overall performance under read-heavy workloads. Therefore, it must be optimized for ROTs, whose performance is generally worse than non-transactional reads due to the coordination overhead to establish a consistent view across shards. To this end, leveraging the knowledge from PORT [58] and insights from other solutions and strategies, we aim to build a reference architecture that manifests the performance-optimal properties of ROTs.

2) **Upgrade the consistency guarantees of existing cloud storage services.**
   Existing cloud storage services offer a wide range of benefits, including high availability, data accessibility, and durability, and also relieve organizations from the intricacies of data replication and reliability. However, most services provide weaker consistency models, such as EC.

   In order to leverage the properties provided by existing storage services, the reference architecture aims to incorporate the literature's findings in a storage-agnostic way that, akin to Bailis *et al.* layered approach, can upgrade the consistency guarantees of the underlying storage service provided that it complies with a set of assumptions.

3) **Enable *value semantics*, thereby facilitating auditing and enabling developers to reason about the system's state and data at a point in time.**
   Finally, the inherent complexity of geo-replicated systems makes it desirable to provide a way to audit the system, which can be achieved by implementing the reference architecture with *value semantics*, where system states can be perceived at a given time.

4.4 Research Questions

In order to validate the hypothesis and accomplish the goals established in section 1.4, the following research questions were formulated to guide this work:

RQ1. What are the ideal properties of a geo-replicated causally consistent read-heavy system?
The prevalence of reads in real-world applications indicates the significance of optimizing the latency and throughput of ROTs for improving the system's overall performance and meeting customer expectations and service requirements. In this regard, reviewing the results achieved in PORT [58], particularly the performance-optimal properties of ROTs, is crucial for understanding how a read-heavy system can be designed to ensure the desired performance.

RQ2. What properties and strategies do existing causally consistent systems employ?
In order to build a reference architecture for read-heavy systems, it is crucial to review the architectural and algorithmic properties of existing systems, analyze their trade-offs, and determine how their strategies for ensuring CC affect the performance-optimal properties of ROTs identified in **RQ1**.

RQ3. What metrics have been used to evaluate these systems?
Given that any reference architecture must be empirically validated and evaluated to prove its applicability, it is relevant to identify which metrics must be considered when assessing the implementation of distributed causally consistent systems.

RQ4. Can we produce a cloud-native reference architecture for read-heavy systems that manifests the performance-optimal properties of read-only transactions? How does it compare with state-of-the-art causally consistent systems?
Drawing upon the review undertaken in **RQ1** and **RQ2**, how can we produce a reference architecture that ensures optimal ROT performance? How does it incorporate the literature's findings, and which trade-offs arise from this integration?

RQ5. How can this reference architecture guarantee CC above eventually consistent cloud storage services? What trade-offs arise from offering these guarantees? Considering the benefits of existing cloud storage infrastructure, particularly their high availability, data accessibility, and durability, we aim to assess how the architecture can upgrade the consistency guarantees of weakly consistent cloud storage systems and to determine the impact of this decision on the metrics identified in **RQ3**.

RQ6. How can we realize this reference architecture to ensure *value semantics*?
Geo-replication and sharding are paramount in large-scale applications to ensure fault tolerance, minimize access latency, and handle large volumes of data. However, they impose an

extra complexity on the system's architecture and communication across replicas to ensure consistency. Consequently, they make it difficult for developers to reason about the system's behavior, understand the flow of data, and identify the root cause of a failure. In this regard, if the reference architecture is designed to ensure *value semantics*, it can provide a way to perceive the system as a succession of atomic states, thus enabling developers to audit the system and reason about its behavior at a point in time.

Table 4.1 maps each of these research questions to the section of the document where it is addressed.

ID	Research question	Related chapters
RQ1	What are the ideal properties of a geo-replicated causally consistent read-heavy system?	Chapters 2 and 3
RQ2	What properties and strategies do existing causally consistent systems employ?	Chapter 3
RQ3	What metrics have been used to evaluate these systems?	Chapter 3
RQ4	Can we produce a cloud-native reference architecture that manifests the ideal properties of a geo-replicated causally consistent read-heavy system? How does it compare with state-of-the-art causally consistent systems?	Chapters 6 to 8
RQ5	How can this reference architecture guarantee CC above eventually consistent cloud storage services? What trade-offs arise from offering these guarantees?	Chapters 6 to 8
RQ6	How can we realize this reference architecture to ensure *value semantics*?	Chapters 7 and 8

Table 4.1: The research questions in this dissertation and where they are addressed in the document.

4.5 Validation and Evaluation

Zelkowitz and Wallace [99] classified experimental models for technology validation into the following four categories:

Scientific method. Scientists develop a theory to explain a phenomenon; they propose a hypothesis and then test alternative variations of the hypothesis. As they do so, they collect data to verify or refute the claims of the hypothesis.

Engineering method. Engineers develop and test a solution to a hypothesis. Based on the results of the test, they improve the solution until it requires no further improvement.

Empirical method. A statistical method is proposed as a means to validate a given hypothesis. Unlike the scientific method, there may not be a formal model or theory describing the hypothesis. Data is collected to verify the hypothesis.

Analytical method. A formal theory is developed, and results derived from that theory can be compared with empirical observations. [99].

The validation of the present dissertation uses both Engineering and Empirical methods.

In the first phase, the knowledge extracted from the literature review is translated into four system architectures, which serve as the starting point for the development of the reference architecture. Following an engineering approach, the reference architecture is then iteratively built and refined through the Architectural Tradeoff Analysis Method (ATAM) method, illustrated in fig. 4.1. This technique resorts to a spiral model that enables an iterative evaluation of architecture-level designs with respect to a set of quality attributes. This process culminates in three candidate architectures. These architectures are then evaluated against ten scenarios, each targeting one of the selected quality attributes: performance, data staleness, and consistency. Following the scenario realization and analysis, we discuss the trade-offs of each architecture and select the one that offers greater scalability and adaptability to existing storage services.

In the second phase, the reference architecture is realized in a prototype system, using Amazon S3 as the underlying storage service. The prototype is deployed using Amazon Web Services (AWS)'s cloud infrastructure and evaluated against a set of metrics: read latency and throughput, data staleness, and goodput. We compare the prototype against its base system, Amazon S3, and assess the trade-offs of upgrading its consistency guarantees with CC. Additionally, to empirically validate the suitability of the prototype and its underlying architecture to read-heavy systems, we evaluate the latency and throughput of the system under read-heavy workloads and assess how it scales with the number of nodes and clients. Finally, we experimentally show how the system's state can be reconstructed through its logs, enabling the developer to reason about its behavior at a point in time or to observe the progression of values of a particular key.

4.6 Summary

Section 4.1 outlined the shortcomings identified in the literature review, particularly the lack of general solutions for applying CC to real-world read-heavy systems. It also highlighted the scarce evidence on how existing solutions perform when realized in real-world systems, the need to integrate the benefits offered by cloud storage services, and the potential for improvement in terms of auditability. Finally, it pointed out the inexistence of a system that achieves all the performance-optimal properties of ROTs and supports partial replication.

Section 4.2 described the scope of this work — assess how the results achieved in existing causally consistent key-value stores can be extrapolated into a storage-agnostic reference architecture for read-heavy systems. Additionally, this work aspires to make the system auditable.

Then, section 4.3 presented the hypothesis that we aim to validate:

> *There exists a reference architecture that (1) manifests the ideal properties of geo-replicated causally consistent read-heavy systems, (2) upgrades the consistency guarantees of existing cloud storage services, and (3) enables value semantics, thereby*

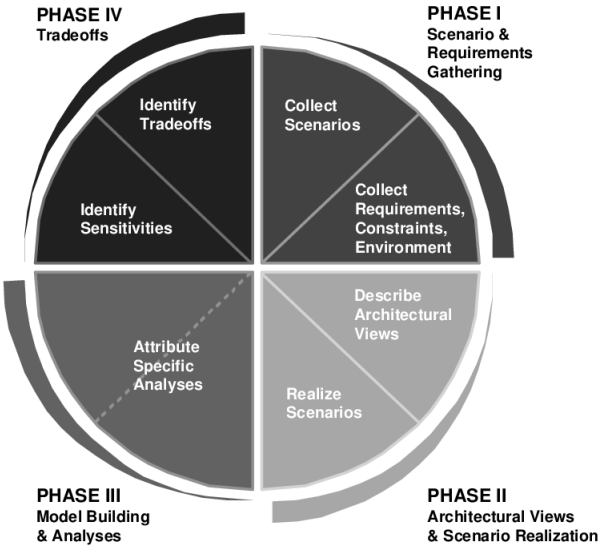


Figure 4.1: Steps of the ATAM [42]. This technique resorts to a spiral model that enables an iterative evaluation and refinement of architecture-level designs concerning some quality attributes. In the first phase, the scenarios and requirements are set. Then, candidate architectural views are proposed and evaluated against each scenario. After the evaluation, the architecture's trade-offs are analyzed.

> *facilitating auditing and enabling developers to reason about the system's state and data at a point in time.*

The main research questions that will drive this work were then outlined in section 4.4.

Finally, section 4.5 described the validation process of this dissertation. First, following an engineering approach [99], the reference architecture is iteratively built and refined through the ATAM method and realized in a prototype system, using Amazon S3 as the underlying storage service. Then, following an empirical method, the prototype system is assessed based on several metrics, namely read latency and throughput, goodput, and data staleness. The prototype is compared against its base system to assess the impact caused by CC and evaluated under read-heavy workloads to study its suitability to read-heavy systems.

Chapter 5

Preliminary Studies

Contents

In the previous chapter, we set out to assess whether the results achieved in existing causally consistent key-value stores could be extrapolated into a reference architecture for read-heavy systems. To that end, before delving into the development of the reference architecture, we felt the need to extend the literature review conducted for **RQ2** by extracting the high-level architectures of some of the systems identified in our review. Leveraging the results from this abstraction process, which we describe in section 5.1, we followed the ATAM methodology to iteratively build and refine a set of candidate architectures. This process, which we describe in section 5.2, resulted in a final reference architecture further detailed in the following chapter. Finally, section 5.3 summarizes the topics above.

5.1 State-of-the-Art Causally Consistent Architectures

Extending the literature review conducted for **RQ2**, the ideation phase of the reference architecture entailed an analysis of the architectures of two state-of-the-art causally consistent systems (PaRiS [85] and Eiger-PORT [58]) and of the two architectural approaches described in section 3.3 (Bolt-on [8] and MongoDB [91]).

From the systems surveyed in chapter 3, PaRiS was the only system to enable server-side partial replication. Furthermore, it was one of the few systems that complied with two of the three *NOC properties*, so it was worth analyzing its architecture to understand how it could be adapted to support the *one-round* property. Bolt-on's architecture takes a layered and storage-agnostic approach to CC, thus constituting a significant reference for designing the proposed reference architecture to support existing cloud storage services. MongoDB was one of the first industry databases to provide CC, presenting a proven solution for efficiently implementing CC at scale.

Finally, Eiger-PORT [58] was specially designed to guarantee the *NOC properties*, and thus, deriving its architecture provides a baseline for comparison with the remaining architectures.

Through our analysis, we identified the core architectural characteristics and building blocks of these systems and translated them into the following high-level architectures:

- **PaRiS Architecture.** Based on PaRiS [85], this architecture portrays a stabilization-based system, where any server can assume the role of coordinator or cohort, and CC is enabled through a stabilization protocol and a client-side cache.
- **Bolt-on Architecture.** This architecture represents Bolt-on's [8] layered approach where a shim layer upgrades an ECDS with CC.
- **MongoDB Architecture.** Based on MongoDB's [91] redesign, this architecture illustrates a system where writes are sequenced in a single node per shard and replicated to secondary nodes that only support read operations.
- **Eiger-PORT Architecture.** Based on Eiger-PORT [58], this architecture can support performance-optimal ROTs by letting clients define the snapshot from which to read.

5.1.1 PaRiS Architecture

PaRiS [85] is a partially replicated key-value store that provides all but one of the *NOC properties* — it uses a single timestamp to track dependencies, provides non-blocking ROTs, but it uses a coordinator-based approach which requires two communication rounds.

PaRiS uses a stabilization protocol (UST) to ensure updates are only made visible when they are stable across all DCs. Additionally, it uses a client-side cache to store updates not yet reflected in the snapshot defined by the UST. PaRiS uses a single scalar timestamp for tracking dependencies and a HLC to timestamp events.

The diagram in fig. 5.1 illustrates the high-level architecture derived from PaRiS's system description. The client side provides an interface for performing generic read/write transactions and a private cache. Servers may assume the role of coordinator or cohort for different transactions. In order to handle a transaction, coordinator servers may need to communicate with other partitions (cohorts) from the local DC and possibly from remote DCs. For that purpose, servers need to accept connections from other servers to read a set of keys and to collaborate in the 2PC protocol, which is necessary for write transactions.

Periodically, each server is also responsible for persisting committed updates into its copy of the key-value store and replicating updates to other replicas, which in turn must be able to receive replication requests and apply them to their local key-value stores.

Finally, given that any node can be a coordinator and that clients may read from partitions in other DCs, servers across all DCs must asynchronously determine the stable frontier (i.e., a timestamp below which all servers have received all updates). Figure 5.1 illustrates PaRiS UST protocol, which organizes the servers in a tree to compute the stable frontier efficiently.

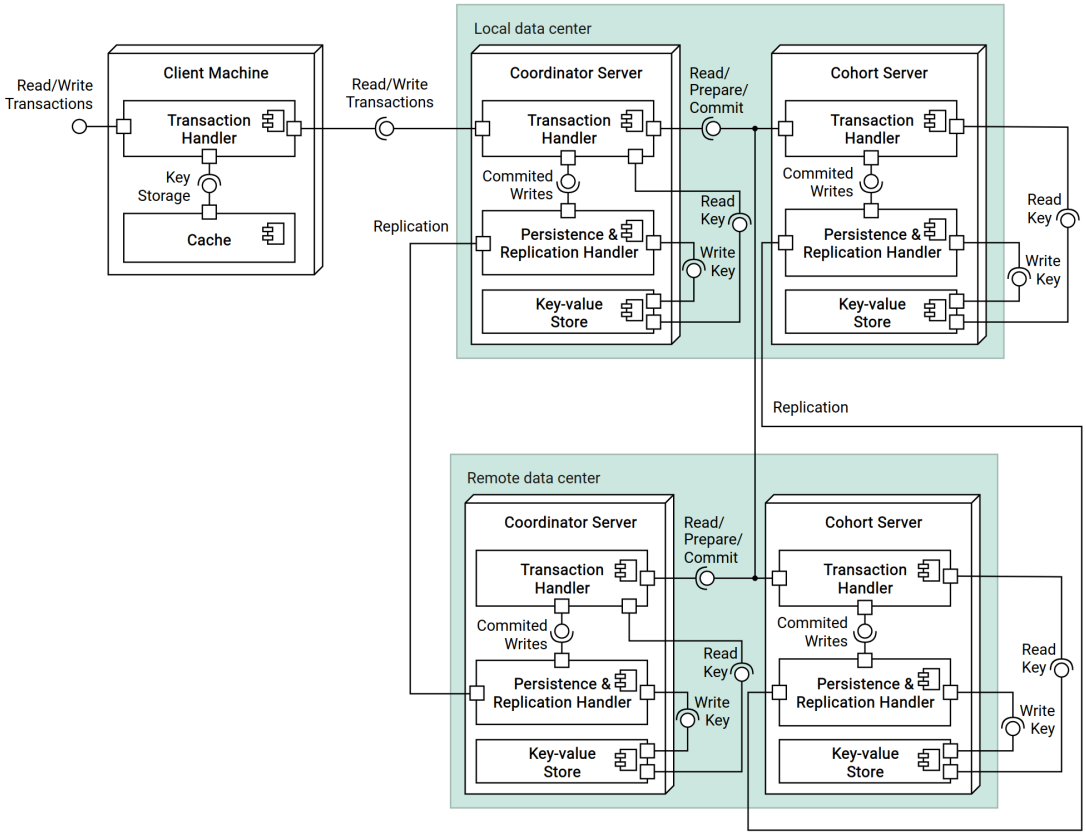


Figure 5.1: PaRiS architecture UML deployment/component diagram. The client-side has a library that forwards the requests to the server and a cache that ensures read-your-writes. Servers adopt a coordinator-based approach: the coordinator partition communicates with cohort partitions of the same or other DCs to request versions of keys that it does not hold and perform the 2PC protocol. Persistence and replication are handled asynchronously (Based on PaRiS [85]).

Considering PaRiS architecture, we conclude that, to avoid the two communication rounds required to perform a ROT, a client would need to directly contact every involved partition, providing it with the transaction's snapshot. Therefore, it would need to be capable of determining a snapshot that is stable in every partition, even before long periods of inactivity or when it joins the system.

5.1.2 Bolt-on Architecture

As described in section 3.3, Bailis *et al.* [8] proposed a layered approach to CC that separates the safety property of data consistency from the architectural concerns of liveness, replication, and durability. Their proposed architecture upgrades an ECDS through a shim layer that provides CC. Thus, it represents the first direction toward standardizing an architecture for causally consistent systems.

Bolt-on's approach uses vector clocks to track dependencies. To ensure that the shim is always causally consistent, it only makes a write visible to a client when that write forms a causal cut with

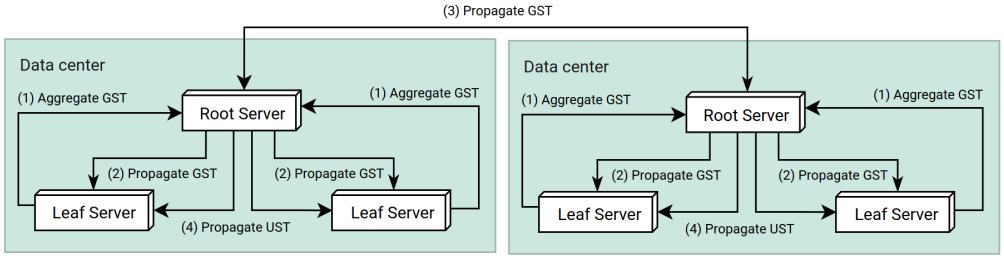


Figure 5.2: UST Stabilization Protocol implemented by PaRiS [85] to identify stable snapshots. Nodes within a DC are organized in a tree. Periodically, partitions within a DC exchange the minimum timestamp they have seen from all replicas. The minimum of the exchanged values (GST) identifies a timestamp below which all transactions were applied in all partitions of the DC. Root servers propagate the GST back to the leaf servers and exchange it with the root servers of other DCs. Each root server calculates the UST as the aggregate minimum of the GSTs seen from all DCs. The UST identifies a timestamp below which all partitions in every DC have applied all transactions with lower timestamps (Based on PaRiS [85]).

the current local store.

The diagrams in fig. 5.3 and fig. 5.4 illustrate the two architectural approaches proposed in this work.

In the optimistic approach (fig. 5.3), clients contact a shim, which works as a client-side library, to perform non-transactional read and write operations. The shim handles the operations by reading or writing in a local store. A resolver process asynchronously updates the local store by fetching the most recently accessed keys from the ECDS.

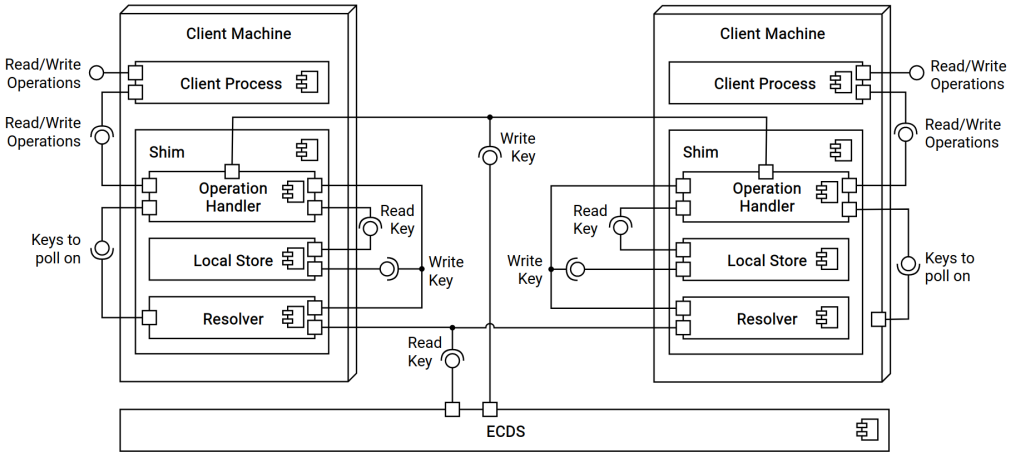


Figure 5.3: Bolt-on optimistic architecture UML deployment/component diagram. Clients contact a shim component, which works as a client-side library, to perform non-transactional read and write operations. The shim handles the operations by reading or writing in a local store. A resolver process asynchronously updates the local store by fetching the last accessed keys from the ECDS (Based on Bolt-on [8]).

In the pessimist approach (fig. 5.4), the shim tries to get the most recent version of the requested keys by synchronously and repeatedly reading from the ECDS and updating the local store until all missing dependencies are satisfied.

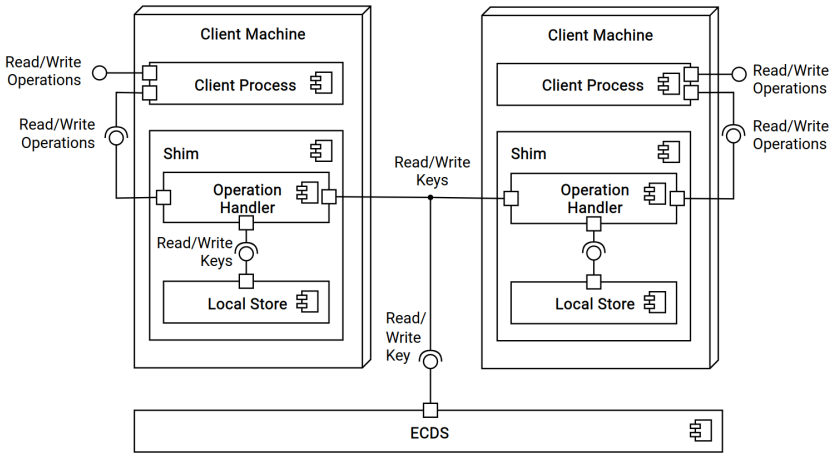


Figure 5.4: Bolt-on pessimistic architecture UML deployment/component diagram. Writes are immediately applied in the local store and persisted to the ECDS. Upon a read request, the shim component synchronously fetches the most recent version of the requested keys, recursively gets any missing dependencies, and updates its local store (Based on Bolt-on [8]).

A vital benefit of both approaches is their modularity, separation of concerns, and ability to support modern cloud storage services, relieving software engineers from the intricacies of data replication and reliability and increasing data accessibility. If combined with novel ideas from the research community, this approach could be extended with ROTs and leverage other dependency tracking forms to reduce the metadata overhead.

5.1.3 MongoDB Architecture

In 2019, Tyulenev *et al.* [91] shared the design choices that drove the implementation of CC in MongoDB, a well-known distributed database that supports replication and sharding. Given the ubiquity of MongoDB and its consequent performance and scalability requirements, their contribution is valuable for understanding the possible paths for providing CC.

MongoDB uses HLCs to timestamp events, a single timestamp to track dependencies, and synchronizes clocks between shards by advancing the clock.

In MongoDB's architecture (illustrated in fig. 5.5), each shard comprises one primary node and several secondary nodes. Primary nodes may accept both write and read operations, while secondaries are read-only nodes, which only receive updates from the primary node. Primary storage nodes handle writes by adding them to the operation log and answer read requests by reading directly from memory. Write nodes behave like state machines that apply the operations from the log to the data set. The log is replicated to the secondary nodes of the same shard, which follow a similar behavior, applying the updates to their data sets.

Clients use a driver to dispatch read and write requests. The driver, in turn, connects to a query router, which uses the configuration servers to fetch the configuration and route the requests to the appropriate primary nodes.

Overall, MongoDB's architecture offers the ability to distribute read load across secondary nodes and can easily scale horizontally. These features may prove particularly valuable in read-heavy systems.

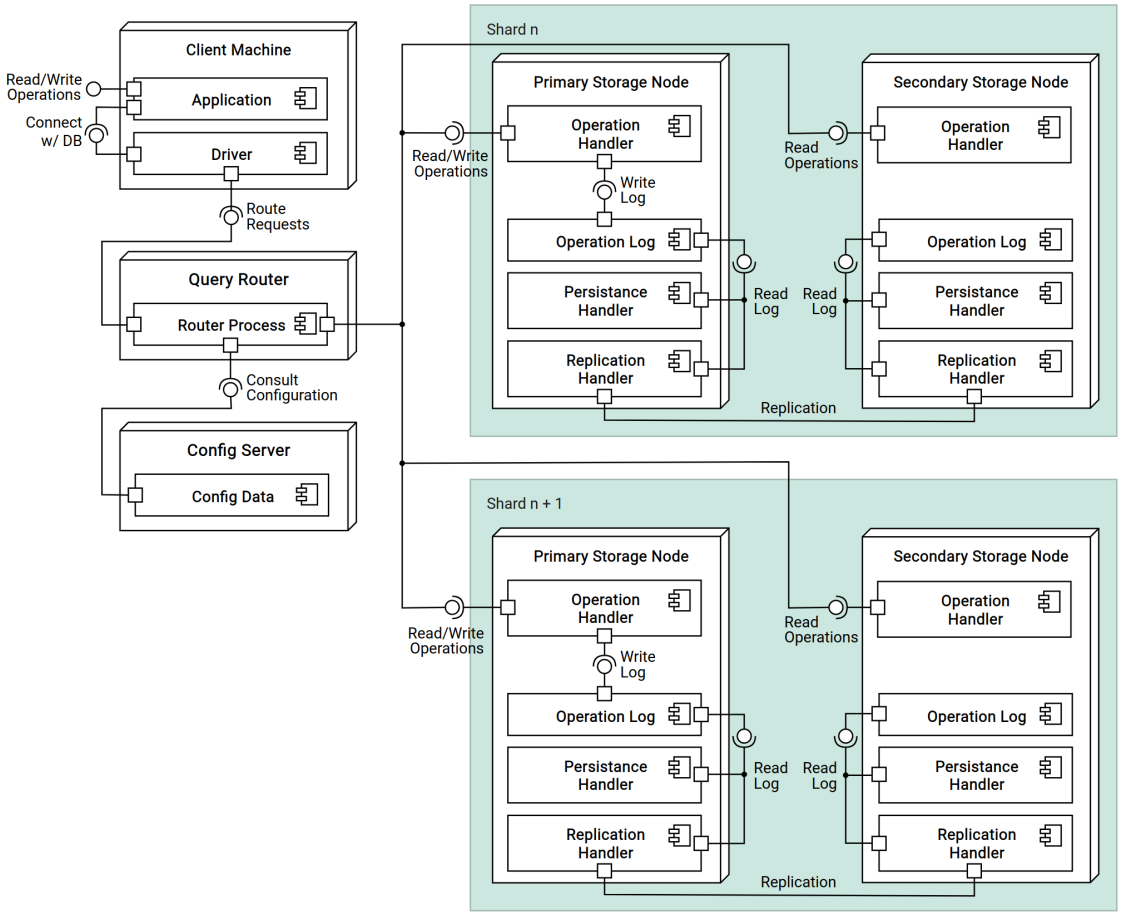


Figure 5.5: MongoDB architecture UML deployment/component diagram. A driver dispatches read and write requests to a query router, which uses the configuration servers to route the requests to the appropriate primary nodes. Each shard comprises one primary node and several secondary nodes. Primary nodes accept write and read operations, while secondaries are read-only nodes. Primary nodes register the operations in an operation log, which is periodically persisted and propagated to the secondaries (Based on MongoDB [91]).

5.1.4 Eiger-PORT Architecture

PORT [58] is a system design that provides optimal ROT performance with the best possible consistency guarantees. When applied to Eiger's [54] causally consistent system, it preserves its consistency properties and upgrades its transactional protocol to ensure the three *NOC properties*.

PORT's design relies on version clocks to capture causality: the servers track the last version-stamp of their latest write, and the client tracks the minimum of such versionstamps he has seen from the servers (the stable frontier). This value is used to perform subsequent requests.

Figures 5.6 and 5.7 illustrate the PORT's design when realized in Eiger. Similarly to PaRiS architecture, a client-side library accepts transactions. However, Eiger-PORT does not support generic transactions and does not require a coordinator server for ROTs.

As we can observe in fig. 5.6, each server accepts ROT requests from clients, which it performs from a multi-versioning framework by indexing the desired key with the client-provided timestamp. This approach eliminates the need for inter-server communication, requires a single scalar timestamp to be transferred to each involved server, and never blocks because it enables stale reads.

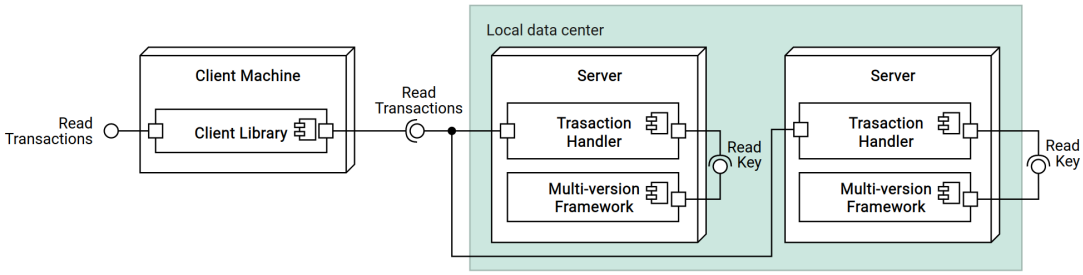


Figure 5.6: Eiger-PORT architecture UML deployment/component diagram (ROTs). All servers accept Read-only transaction (ROT) requests from clients. To handle a ROT request, the server indexes the version chain of the desired key using the client-provided timestamp (Based on Eiger-PORT [58]).

As shown in fig. 5.7, Eiger-PORT's servers also accept WOT requests. Like in PaRiS, servers may assume the role of coordinator or cohort in the WOT protocol. However, the client interacts directly with the coordinator and the cohorts to deliver the transaction's snapshot. All servers accept connections from other servers to collaborate in the 2PC protocol. Replication occurs asynchronously to remote DCs.

Eiger-PORT's architecture enables all *NOC properties* because, in contrast with most coordinator-based approaches where the coordinator determines a snapshot that is stable across partitions or even DCs, it uses a client-provided timestamp to perform the transaction. This architectural choice may penalize staleness, especially when dealing with high inter-operation delays.

5.1.5 Summary

In PaRiS [85], MongoDB[91], and Eiger-PORT [58], the client process must attach a timestamp to the requests, which will be used to determine the snapshot of the transaction. In MongoDB, the driver component provides that functionality, while in PaRiS and Eiger-PORT, the last update timestamp is stored in the client's state. In both Bolt-on architectures [8], the client side provides more complex functionality, given that causality is exclusively guaranteed by the client-side shim

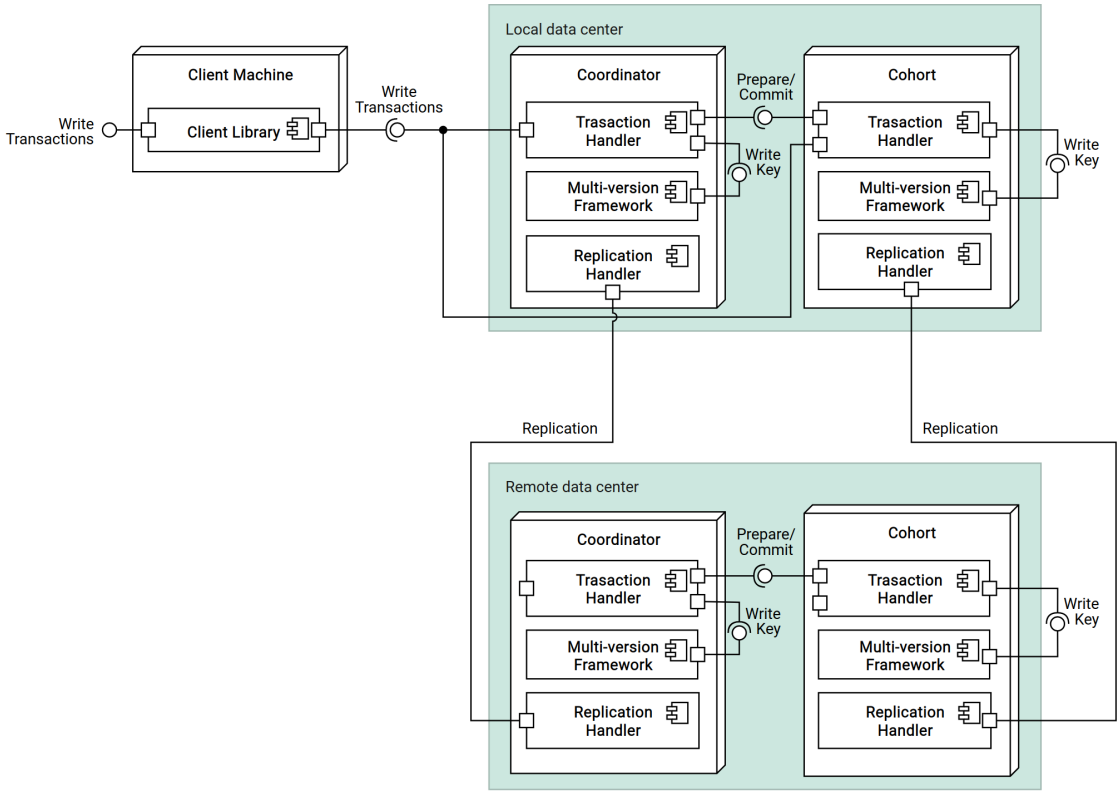


Figure 5.7: Eiger-PORT architecture UML deployment/component diagram (WOTs). Eiger-PORT's Write-only transaction (WOT) protocol adopts a variant of the 2PC protocol where clients interact with all involved servers to deliver the transaction's snapshot. Replication occurs asynchronously to remote DCs (Based on Eiger-PORT [58]).

layer. Like in MongoDB, PaRiS, and Eiger-PORT, clients must track their causal history. However, Bolt-on uses explicit dependency tracking, while other approaches require a single timestamp.

PaRiS's and Eiger-PORT's servers support transactions (PaRiS enables generic transactions while Eiger-PORT provides ROTs and WOTs). MongoDB and Bolt-on only support non-transactional reads and writes. In PaRiS, ROTs are sent to a single server that determines the transaction's snapshot and then contacts the necessary servers, resulting in two rounds of communication. In contrast, in Eiger-PORT's architecture, requests are directly forwarded to all involved servers, supplying them with the snapshot for the ROT. In MongoDB, a query router contacts all the necessary partitions.

Like in PaRiS, Eiger-PORT uses the 2PC protocol to perform WOTs. However, in PaRiS, the client only contacts the coordinator node, while in Eiger-PORT, the client connects directly with every partition involved in the transaction.

PaRiS and Eiger-PORT use a multi-master architecture, whereas MongoDB uses a primary-replica strategy where only the primary node of each shard may perform write operations.

PaRiS, Eiger-PORT, and MongoDB are multi-versioned, whereas Bolt-on is built on top of an

ECDS that provides single register semantics. In all systems, a local data store keeps a copy of the data either in the servers or, in the case of Bolt-on, in a client-side local store.

Except in Bolt-on's pessimistic approach, persistence is handled asynchronously.

Regarding synchronization, PaRiS uses a gossiping protocol and heartbeats (in the absence of updates) to track dependencies, whereas, in MongoDB, servers only advance the clock when clients make requests with higher logical times than the server. It does not require a stabilization protocol because only a single node sequences the writes and propagates them to secondaries. In Eiger-PORT, given that there is no stabilization protocol, the freshness is affected by the inter-operation delay of the clients. In PaRiS, the client cache ensures read-your-writes, and in MongoDB, the local data store is always guaranteed to be causally consistent. In MongoDB, it may be necessary to wait if reading from a secondary until the requested keys are replicated.

To sum up, PaRiS and Eiger-PORT present a multi-master architecture, enabling load distribution across all servers. While Eiger-PORT ensures performance-optimal ROTs, this comes with the cost of increased staleness, mainly when dealing with low request frequencies. On the other hand, PaRiS architecture requires an extra communication round but supports partial replication and keeps staleness bounded by using heartbeats in the absence of updates to ensure the progress of the UST. Despite the potential metadata overhead, Bolt-on's architecture stands out for its storage-agnostic nature, providing notable advantages compared to alternative approaches that do not leverage existing cloud storage services. Lastly, MongoDB presents a log-based strategy, enabling greater auditability. Additionally, its primary-replica strategy enables balancing read load across secondary nodes. Finally, MongoDB does not require a stabilization protocol despite using a similar strategy as PaRiS to track causality.

5.2 Architecture Trade-off Analysis

Based on the knowledge acquired from the literature review and the abstraction process described in the previous section, the development of the reference architecture used the ATAM methodology. This structured technique enables an iterative refinement and evaluation of architecture-level designs concerning some quality attributes.

This section describes the architecture trade-off analysis of three candidate architecture-level designs. First, section 5.2.1 identifies the quality attributes the architectures will be assessed against. Then section 5.2.2 outlines the requirements considered when designing the candidate architectures, and section 5.2.3 lists the ten scenarios that guided the analysis. Section 5.2.4 describes three candidate architectures that resulted from iterative refinement. Finally, section 5.2.5 assesses the architectural views against each scenario, and section 5.2.6 summarizes the main trade-offs identified throughout the analysis.

It is worth mentioning that, even though the following sections describe the evaluation linearly, the evaluation process required an iterative refinement of the architectural views and scenarios and a re-evaluation of the different views against these scenarios. For simplicity, we only include the analysis of the three final candidate architectures.

5.2.1 Quality attributes

In this dissertation, we set out to design a causally consistent reference architecture for read-heavy systems. In this regard, the first attribute considered in the evaluation was data consistency. Moreover, considering the relevance of read latency and throughput in read-heavy systems, and the impact of response time on user engagement and satisfaction, we also analyzed a set of performance-related scenarios. However, we leave the actual performance testing for the validation stage of this dissertation (chapter 8), focusing instead on the variables that may affect the latency of each operation. Finally, considering the trade-offs uncovered in the literature review, the candidate architectures were also analyzed with respect to staleness, a property often sacrificed in favor of enhanced performance.

5.2.2 Requirements

Considering our hypothesis, we derived the following requirements (Rs) to guide the design process of our architecture:

Consistency Requirements:

R1. The architecture must provide CC+.

R2. The architecture must support ROTs.

R3. The architecture must be able to upgrade the consistency guarantees of ECDSs.

Data Staleness Requirements:

R4. Data staleness must be bounded (i.e., in the absence of updates to a partition, the system must ensure progress so that clients do not observe arbitrarily stale data).

We did not establish any requirement for the performance quality attribute, given that this initial analysis does not aim to evaluate the architecture empirically but to help refine a set of alternatives to reach a final reference architecture.

These requirements constrained the initial design space, helped define the scenarios of the following section, and guided the refinement of the architecture.

5.2.3 Test Scenarios

For each quality attribute, we then defined a set of scenarios (Ss) that guided the design of the candidate architectures and enabled us to assess how they would behave.

Consistency Scenarios:

S1. A client issues two consecutive writes for items X and Y and another client issues a ROT for the same items.

S2. Clients in different regions concurrently issue two updates to the same item.

S3. A client issues two consecutive writes for different items followed by a ROT for those items.

Performance Scenarios:

S4. A client issues a ROT and receives its response.

S5. The ROT request load increases by a factor of 10.

S6. A client issues a non-transactional write and receives its response.

S7. The write request load increases by a factor of 10.

S8. Multiple concurrent ROTs are issued by different clients.

Data Staleness Scenarios:

S9. A single write request updates a data item, eventually becoming visible to all reading clients.

S10. Updates stop being issued to one partition.

These scenarios served as a way to exercise the candidate architectures, assess their behavior and trade-offs, and get insight into whether the realized architecture could meet its requirements. In particular, scenarios **S1** to **S3** enabled us to make an initial assessment of the architectures' consistency guarantees and thus provided insight into whether they would meet requirements **R1** to **R3**. Scenarios **S4** to **S8** allowed us to identify the factors influencing the requests' latency and the system's throughput and assess the architecture's ability to support concurrent reads. Thus, they provided feedback regarding the performance that it could provide, even though we did not establish requirements for this attribute. The last scenarios provided insight into the architecture's ability to meet requirement **R4** by understanding which factors impact staleness.

5.2.4 Architectural Views

The requirements and scenarios identified in the previous sections led to an initial set of architectural views. These views were iteratively refined and combined, culminating into three candidate architectures (CAs), each presenting a small mutation concerning the previous one.

The three candidate architectures, illustrated in figs. 5.8 to 5.10, are organized into three tiers: the client tier, where a client-side library provides an interface for user requests and forwards them to the servers; the computing tier, which processes the client requests and guarantees CC; and the data tier, a weakly consistent data store that handles data replication and guarantees durability.

The architectures comprise the same physical entities: client machines, write nodes, read nodes, and a data store. The client machines include a client-side library, which exposes two

operations: non-transactional writes, which enable the client to set the value of a specific data item, and ROTs, which allow clients to retrieve a set of data items. The compute tier includes two node types: write nodes, which provide an interface for write operations, and read nodes, which support ROTs. The data tier consists of a general-purpose data store that offers weak consistency.

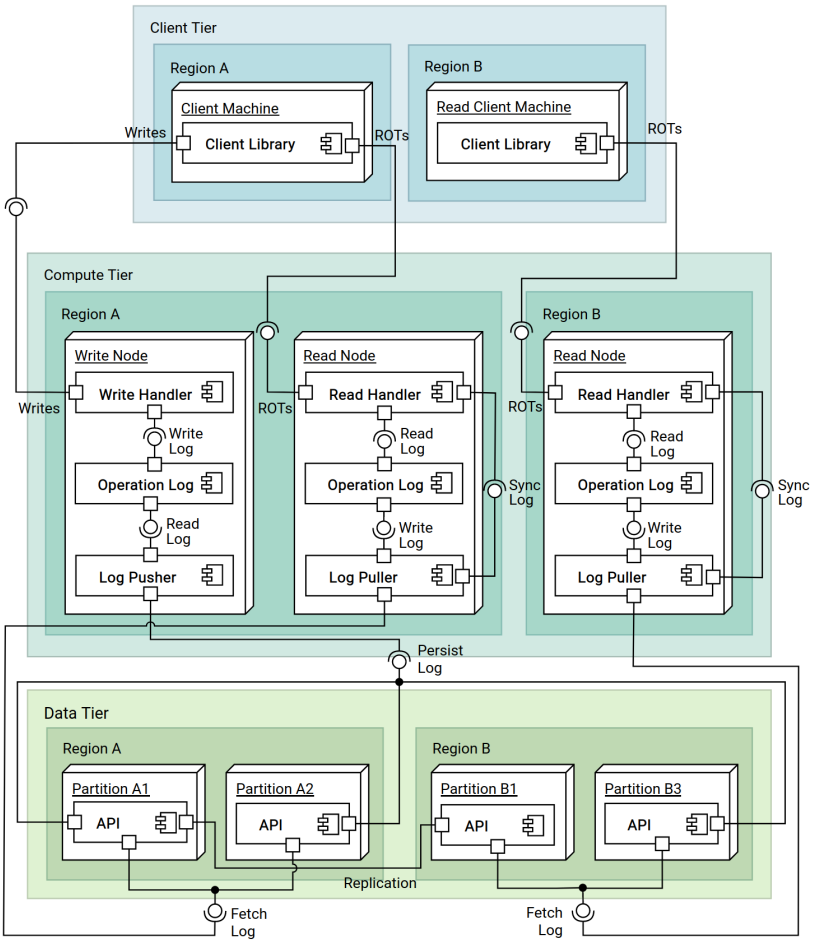


Figure 5.8: Candidate architecture 1 UML deployment/component diagram. This architecture assumes the data store guarantees strongly consistent new writes. Write requests are sequenced in an operation log by a single write node. The log is synchronously persisted in the data store upon a request. Read nodes synchronously fetch the most recent log upon a ROT. The architecture also supports a log per partition.

Moreover, all architectures adopt a log-based approach where write nodes sequence write operations either in a single log or using one log per data partition. They persist the logs through a Log Pusher component in the data store, versioning each log with its clock value. The data store handles replication to other regions. Read nodes use a Log Puller component to fetch the most recent log from the data store, either synchronously or asynchronously.

All architectures assume the usage of HLCs for sequencing write operations.

CA1 - Candidate architecture 1. The first candidate architecture (illustrated in fig. 5.8) assumes that the data store guarantees strongly consistent new writes. This decision restricts its applicability to existing cloud storage services, some of which exhibit eventually consistent behavior. On the other hand, it ensures that read requests have access to the last version of the data. The architecture requires operations to be ordered on a single write node, creating a single point of failure and contention. It also requires the log to be persisted and fetched synchronously upon each request, which may affect latency. However, it does not depend on client-side metadata and minimizes staleness by performing synchronous reads.

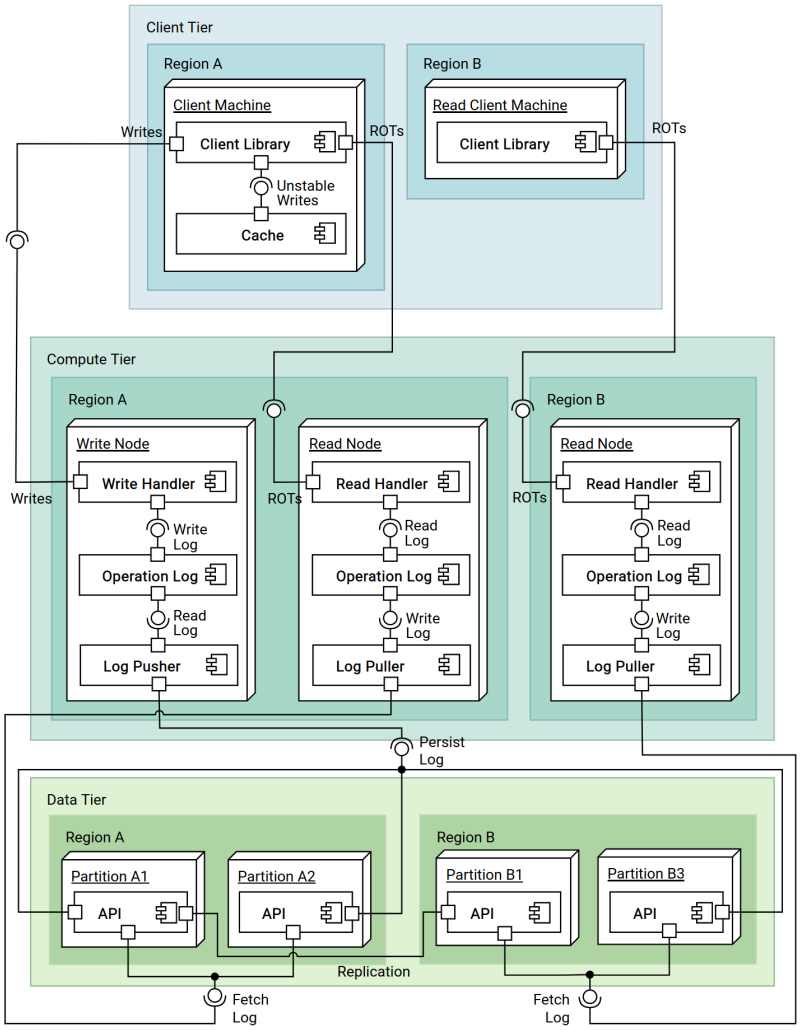


Figure 5.9: Candidate architecture 2 UML deployment/component diagram. A single write node sequences the writes in a log or uses a log per partition and asynchronously persists the logs with the timestamp of the last write. Assuming an ECDS, clients must keep their writes in the cache until they are stable. Read nodes asynchronously fetch monotonically increasing versions of the log and perform ROTs from their local copy of the data.

CA2 - Candidate architecture 2. The second architecture (illustrated in fig. 5.9) removes the data store's behavior restriction, making it more adaptable to eventually consistent cloud storage services. To do so, it adds a client-side cache that ensures read-your-writes. Compared to the previous architecture, it allows write and read nodes to asynchronously persist and fetch the log, trading visibility for improved performance, especially of read requests. This strategy also requires read nodes to keep track of the last log version they saw (`stableTime`) and return it to clients upon a read request so that they can prune their cache. In the case of a log per partition, the `stableTime` is the highest timestamp that is stable across all partitions (i.e., the minimum of the last log timestamps seen from each partition) and only versions whose timestamp is at most as high as the `stableTime` can be made visible.

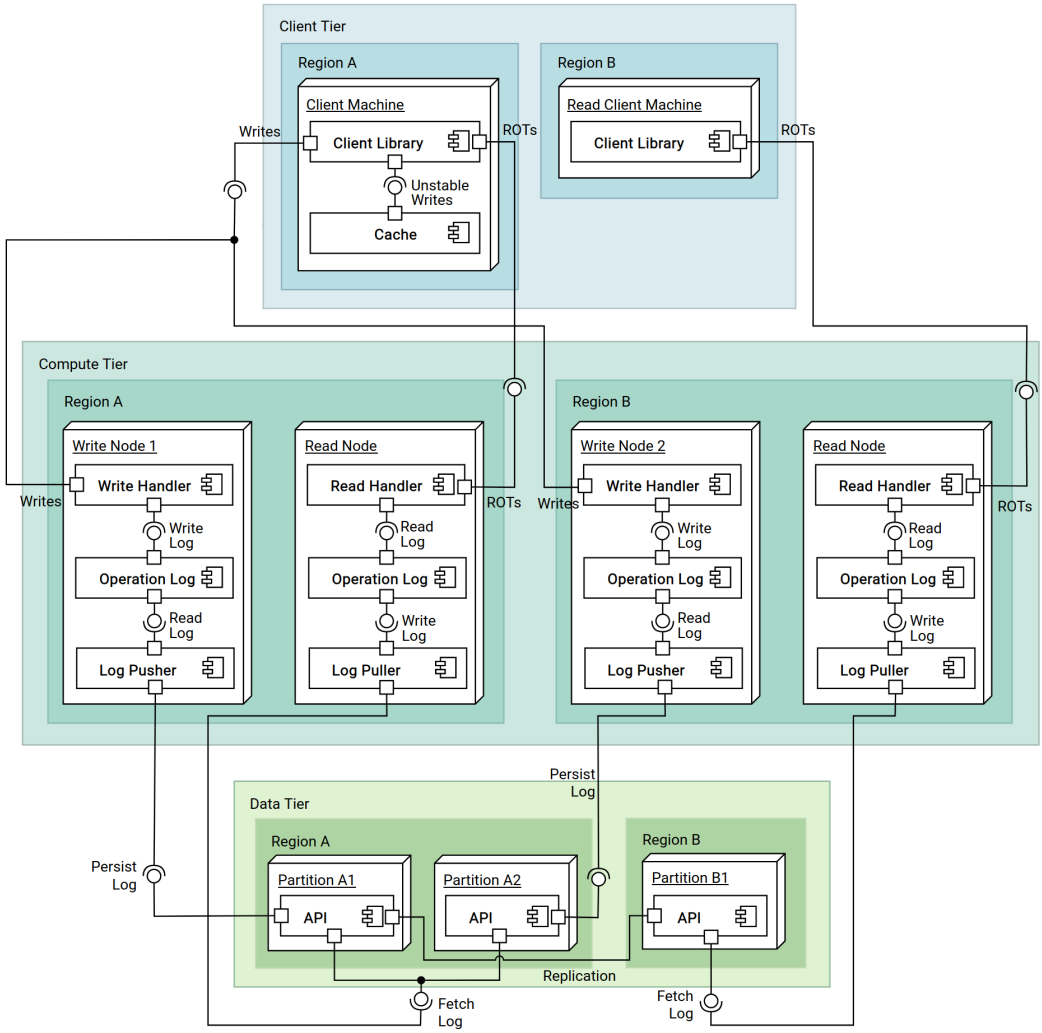


Figure 5.10: Candidate architecture 3 UML deployment/component diagram. Clients forward requests to read and write nodes through a client library. A write node sequences write operations for a given shard in its operation log. The log is asynchronously persisted in the data store. Read nodes periodically fetch the latest log for each partition of their region. Clients store unstable writes in their cache.

CA3 - Candidate architecture 3. In contrast with the previous architectural views, the third architecture (illustrated in fig. 5.10) dedicates a write node per shard, enabling a natural distribution of the write load without the complexity of log merging. Each write node individually sequences operations for a given shard. Like the previous candidate architecture, it also requires a client-side cache. Moreover, given that there is a log per partition, the `stableTime` is the highest timestamp that has been seen from all partitions, and only versions with a timestamp at most as high as the `stableTime` can be made visible. Additionally, clients must track the timestamp of their last write to ensure monotonic writes. Finally, this architecture makes the visibility of new updates to a partition dependent on the frequency of updates in the partitions of the same region. In this regard, it is necessary to establish a synchronization strategy that enables progress in the absence of updates.

5.2.5 Scenario realization

To evaluate the architectural views against the quality attributes, we mapped the scenarios identified in section 5.2.3 onto each candidate architecture.

During this evaluation, we considered an execution environment where a server processes a single request from its input buffer at a time. Even though, in practice, read nodes can concurrently handle multiple ROTs, assuming that each server uses a single thread to process client requests simplifies the analysis for most scenarios.

Finally, for each scenario, we describe the behavior of the three candidate architectures. Given that the three architectures share several similarities, we perform a joint analysis of all candidate architectures for each scenario:

S1. A client issues two consecutive writes for items X and Y, and another client issues a ROT for the same items. In **CA1**, when client C_1 writes to X and then Y, the write node will timestamp Y with a higher timestamp than X. Moreover, as new writes are strongly consistent, the log with the update to Y will only be persisted after receiving the confirmation that the previous log with X's update was successfully persisted and replicated. Read nodes will read from the data store the latest log (or, in the case of a log per partition, the latest log from each partition). Therefore, depending on the time when client C_2 issues the ROT, he will observe one of these outputs: (X_{old}, Y_{old}), (X_{new}, Y_{old}), (X_{new}, Y_{new}).

On the other hand, in **CA2** and **CA3**, the data store does not guarantee that X will be replicated before Y. However, in **CA2**, writes are sequenced by a single write node in order of arrival. Thus, in the case of a single log, Y's write will never be seen before X's because the read node always fetches the most recent log, which will either include both versions or only X's.

In **CA3** and in the case where **CA2** uses one log per partition, two scenarios may occur:

(1) X and Y belong to the same partition, and thus their writes were persisted in the same log by the same write node. This scenario resembles the one described for **CA2** when using a single log.

(2) X and Y belong to different partitions; thus, they are sequenced in different logs. In the case of **CA2**, the same write node sequences both writes in arrival order. In **CA3**, the write node that performs X's write sends back its timestamp, which the client sends in the write request to the write node of Y, ensuring Y's timestamp is higher than X's. Then, both in **CA2** and **CA3**, the read node only makes a version visible if its timestamp is lower or equal to the `stableTime`. Therefore, even if it sees the log from Y's partition before the one from X's, a ROT will never make Y's update visible before making X's.

S2. Clients in different regions concurrently issue two updates to the same item. In the case of **CA1** and **CA2**, updates are totally ordered by the write node. In **CA1**, they will also be seen by read nodes in the same order due to the strong consistency guarantees the data store provides for new writes. In contrast, in **CA2**, if two updates target the same item, each client will keep his write in the cache to ensure read-your-writes. Therefore, clients can observe concurrent requests in a different order. If the client whose write got the lowest timestamp keeps reading the same data item, it will eventually observe the value of the second write when it becomes stable. In **CA3**, there is a log per partition. Therefore, when multiple clients update the same item, they will be handled by the same write node, leading to similar behavior as described for **CA2**.

S3. A client issues two consecutive writes for different items followed by a ROT for those items. In **CA1**, a single write node timestamps new writes so that the second write will have a higher timestamp than the first. Given that writes are strongly consistent, the second write will only be persisted after the first. Therefore, when issuing the ROT, the client is guaranteed to read his writes or writes that causally succeed his writes because the read node synchronously fetches the last log from the data store upon each request.

In **CA2**, even though the first write will have a lower timestamp than the second, the read node may fetch the second log before the first. Regardless, each log contains the sequence of operations, and a read node always fetches the last log. So it always applies the operations in order and only then makes new stable writes visible. Furthermore, if the read node returns a snapshot that does not yet reflect the client's writes, the client library will return the cache's value.

CA3 behaves like **CA2** in this scenario.

S4. A client issues a ROT and receives its response. Assuming an even distribution of clients across regions, there are C/R clients per region. The worst case occurs when all clients send a ROT to their local read node and is experienced by the client whose request is the last to be answered (assuming a single-threaded execution). Therefore, we have the following ROT latency:

CA1: $C/R*(RTT+t_{Queue}+t_{ReadDS}+t_{ParseLog}+t_{Read}*K)$

CA2 and CA3: $C/R*(RTT+t_{Queue}+t_{Read}*K+t_{PruneCache}+t_{ReadCache}*K)$

In **CA1**, ROT latency includes the round-trip time to the server, the time that a request waits to be processed, and the time to synchronously fetch the logs from the data store, parse them, and

read the desired keys. In **CA2** and **CA3**, the read is processed without contacting the data store (i.e., t_{ReadDS} and $t_{ParseLog}$ are removed). Furthermore, in these architectures, the server's response includes the timestamp that identifies the stable time across partitions, enabling the client to prune his cache. So, read latency also includes the time to prune the cache and read the desired versions ($t_{PruneCache} + t_{Readcache} * K$).

S5. The ROTs request load increases by a factor of 10. In all architectures, assuming a single-threaded execution, the latency of ROTs increases by ten in the worst-case scenario where all requests arrive simultaneously.

S6. A client issues a non-transactional write and receives its response. A single node handles all the writes in **CA1** and **CA2**. Thus, if we have C clients, the worst case occurs when all clients send an update to the write node and is experienced by the client whose request is the last in the queue. In **CA3**, there is one write node per partition, so the worst case occurs if all clients issue a write request for the same partition. If requests are evenly distributed across write nodes, multiple write requests can be handled simultaneously. The architectures enable the following write latency:

CA1: $C * (RTT + t_{Queue} + t_{Write} + t_{SerializeLog} + t_{WriteDS} + t_{Rep})$

CA2: $C * (RTT + t_{Queue} + t_{Write})$

CA3: $C * (RTT + t_{Queue} + t_{Write})$ or $C/P * (RTT + t_{Queue} + t_{Write})$

In **CA1**, the write is performed synchronously to leverage the strong consistency guarantees of the data store, thus taking the time needed to serialize and persist the logs as well as the replication time ($t_{SerializeLog} + t_{WriteDS} + t_{Rep}$). In **CA3**, the writes can be parallelized if targeting different partitions.

S7. The write request load increases by a factor of 10. In **CA1**, if the rate of updates increases by 10, the write latency and log size increase by the same factor in the case of a single log. If using a log per partition, and requests are evenly distributed across partitions, each log increases by $10/P$. Assuming the log is replicated in all R regions, the storage size increases by $10 * R$ items. **CA2** behaves similarly, but as it pushes the log asynchronously, the rate at which the storage size increases will depend on the frequency of log pushes.

In **CA3**, assuming the requests increase evenly across all P partitions, the write latency and log size increase by $10/P$ in each write node. Like **CA2**, the storage growth will depend on the rate of log pushes.

S8. Multiple concurrent ROTs are issued by different clients. All architectural views support concurrent reads. In **CA1**, a worker thread can be used per ROT request to fetch the log, parse it, and return the response to the user. In **CA2** and **CA3**, each worker thread must define the

transaction's snapshot (i.e., the current stable time across partitions), get the most recent versions that belong to that snapshot, and issue the response to the client.

Additionally, all architectures could support multiple read nodes per region. However, that would require clients to be sticky to a read node in **CA2** and **CA3**.

S9. A single write request updates a data item, eventually becoming visible to all reading clients. In **CA1**, the strong consistency guarantees provided for new writes and the synchronous reads from the data store ensure clients always see the latest value. The staleness of the writes will depend on the time to fetch the logs from the data store, parse them and read the requested values ($t_{ReadDS} + t_{ParseLog} + k * t_{Read}$). However, for each ROT, a new request must be made to the data store.

In **CA2** and **CA3**, given that the data store does not offer strongly consistent new writes and that the read nodes read asynchronously and periodically from the data store, the staleness of reads will depend on the propagation delay of the data store, on the periodicity with which the logs are pushed and pulled and on the time to serialize and parse each log.

S10. Updates stop being issued to one partition. In **CA1** and **CA2**, considering a single log, the log is pushed to the data store by the write node as long as there are updates in any partition. Thus, the staleness of the writes to other partitions will not be affected. On the other hand, in **CA3** and in case **CA2** is used with a log per partition, if no synchronization protocol is in place, the staleness will depend on the partition with fewer updates. A synchronization protocol is required to ensure progress in the absence of updates.

5.2.6 Trade-off Identification

Following the scenario realization and analysis, we can observe a trade-off between staleness and performance. We can maximize visibility by synchronously applying and reading new versions from the data store, which will jeopardize the performance of reads and writes. On the other hand, we can have periodic accesses to the data store, abiding frequent synchronous writes and reads but penalizing visibility.

Concerning the adaptability of the candidate architectures, **CA1**'s strong assumptions regarding the storage layer make it inapt for most eventually consistent data stores. On the other hand, both **CA2** and **CA3** require additional logic both on the client to ensure read-your-writes (i.e., the client cache) and on read nodes to compute the `stableTime`.

In comparison with **CA2** and **CA1**, **CA3** provides greater scalability by splitting the write load across multiple write nodes at the cost of increased data staleness. Moreover, in the absence of updates, **CA3** requires that write nodes persist the log with their new clock value to ensure the progress of the `stableTime`.

In **CA2** and **CA1**, having a single log requires read nodes to keep track of the entire data set. On the other hand, in **CA1**, using a log per partition requires the read node to fetch not one but many logs synchronously. Having a log per partition requires more server-side logic to identify

the `stableTime` in **CA2** and **CA3**. However, it enables read nodes to only keep track of a partial set of the data store.

Our analysis indicates that **CA3** offers greater scalability by partitioning the write load across write nodes. It is also more adaptable to eventually consistent storage services and provides a way to balance the trade-off between visibility and performance by adjusting the frequency with which the logs are pushed and fetched from the data store. More importantly, **CA3** subsumes both **CA1** and **CA2**, which led us to select it as our reference architecture.

5.3 Summary

This chapter has presented this work's initial research and contributions. This process aimed to collect additional knowledge regarding the architectural designs of existing causally consistent systems and to build an architecture that responds to our hypothesis by incorporating the acquired knowledge into a reference architecture.

In a first contribution (section 5.1), we extended the literature review conducted for **RQ2** by translating some of the systems and approaches identified in our review into high-level architectures. This analysis resulted in a more practical understanding of the architecture and trade-offs of each system. Most importantly, it provided relevant insights for developing the reference architecture.

In a second phase (section 5.2), leveraging the results from the literature review and the insights provided by the architectural analysis described above, we applied the ATAM methodology to iteratively build, refine and evaluate a set of candidate architectures. During this process, we set the quality attributes that would drive the analysis: consistency, performance, and staleness. We then set the architecture requirements and assessed the behavior of each candidate architecture under ten usage scenarios. This process culminated in the proposed architecture, detailed in the following section.

Chapter 6

A Reference Architecture for Read-Heavy Systems

Contents

The previous chapter outlined the preliminary contributions of this work, covering the study of the architectural designs of some state-of-the-art systems and the architectural trade-off analysis of three candidate architectures. This chapter addresses **RQ4** by presenting the reference architecture that resulted from our initial contributions and comparing it with the works identified in the literature review.

First, section 6.1 describes the design of the reference architecture. Then, section 6.2 compares the reference architecture with several works studied in the literature review. Finally, section 6.3 provides a brief summary of this chapter's topics.

6.1 Reference Architecture

This section presents progress towards providing a general solution for applying CC to real-world read-heavy systems by describing the design of the proposed reference architecture.

First, section 6.1.1 synthesizes the core components and characteristics of the reference architecture. Then, section 6.1.2 justifies some of our design choices, such as the type of clock, the dependency tracking strategy, and clock synchronization. Section 6.1.3 states the assumptions about the underlying data store, client-server communication, and data partitioning. The subsections that follow describe the checkpointing strategy (section 6.1.4), garbage collection (section 6.1.5), and fault tolerance (section 6.1.6).

6.1.1 Architecture Description

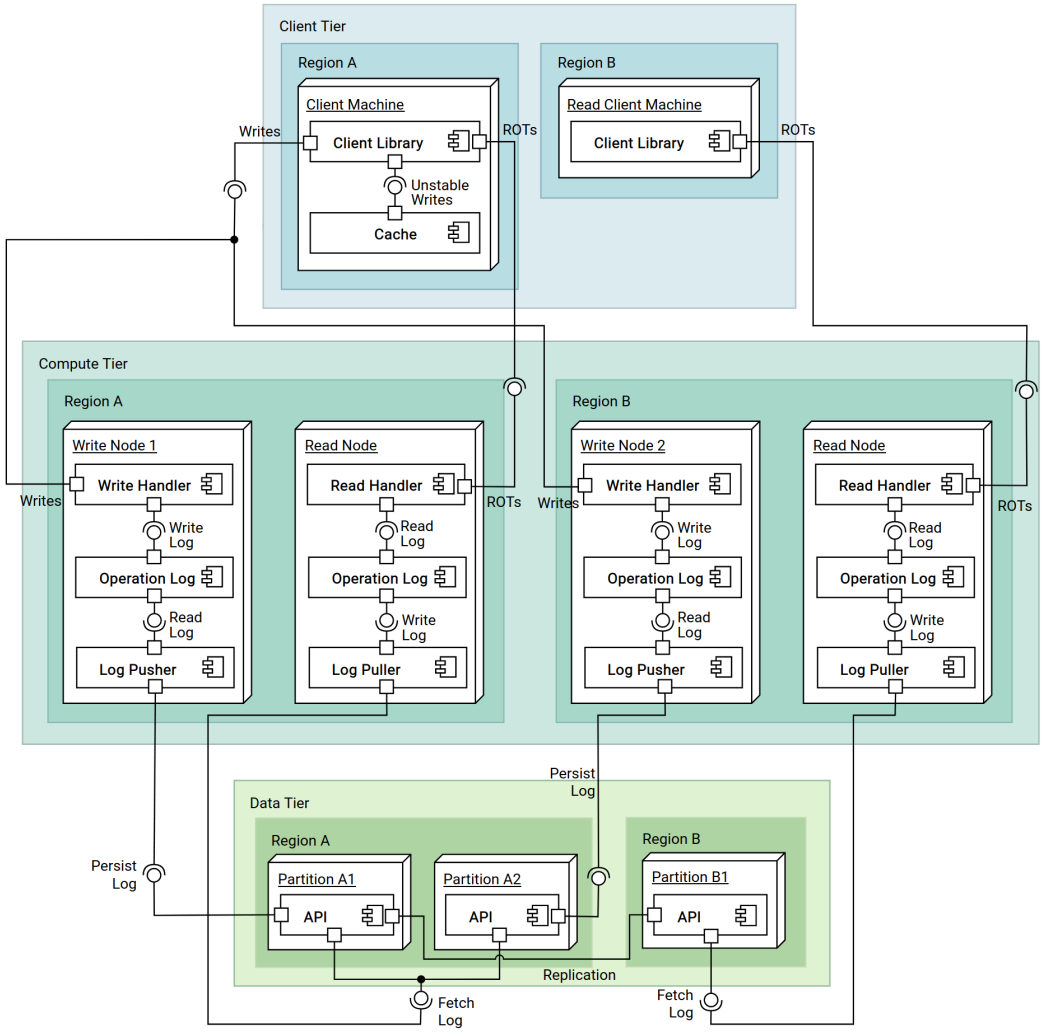


Figure 6.1: Reference architecture UML deployment/component diagram. Clients forward requests to read and write nodes through a client library. A write node sequences write operations for a given shard in its operation log. The log is asynchronously persisted in the data store. Read nodes periodically fetch the latest log for each partition of their region. Clients store unstable writes in their cache.

The proposed reference architecture, illustrated in fig. 6.1, is organized into three tiers: the client tier, where a client-side library provides an interface for user requests and forwards them to the servers; the computing tier, which processes the client requests and possibly implements business rules; and the data tier, which handles data replication and guarantees durability. More importantly, it decouples the implementation of CC, handled by the top and middle layers, from the underlying storage service.

The client tier represents any client-side application or machine that uses the system's library to perform requests to the service. The client-side library interface exposes two operations: non-transactional writes, which enable the client to set the value of a specific data item, and ROTs,

which enable clients to request the values of a set of data items. Furthermore, it attaches the necessary metadata to the requests and abstracts the communication with the compute layer. Each client-side node also comprises a private cache, which stores unstable versions written by the client (i.e., any new update that may not yet be reflected in subsequent reads). The cache is updated upon each write request. The diagram also foresees the existence of write-only or read-only clients, which may be the case in two-sided content-delivery services where content may be produced and consumed by separate entities. In this scenario, clients that exclusively perform read or write operations do not require a private cache to ensure CC.

The compute tier includes two node types: write nodes, which provide an interface for non-transactional write operations, and read nodes, which support ROTs.

Each write node is responsible for a single data partition. Through its HLC, a write node that receives a new write assigns it a hybrid timestamp and registers it in the operation log, which totally orders the writes to that partition. Asynchronously, the Log Pusher persists the log in the data store, timestamping it with the last clock value. To ensure monotonic writes, the client must store the timestamp of his last write (`lastWriteTimestamp`). That timestamp must be sent in the subsequent write request so the server may update its clock and ensure the second write gets a higher timestamp than the first.

Read nodes are responsible for a single region. Clients issue requests to the nearest read node. Read nodes asynchronously and periodically fetch the latest logs from the data store. They must also determine the `stableTime`, the maximum version timestamp they have seen from all the partitions of their respective region. When a read node receives a ROT request, it defines the snapshot of that request as the `stableTime`. It sends the `stableTime` together with the requested values so that the client may prune his cache and determine which values to return to the user to respect causality. If, after pruning, the cache still holds a version of any of the requested items, then the client library must return that version to ensure read-your-writes.

Additionally, in a sharded system where a shard may receive write requests at different rates, write nodes must advance their logical clocks to ensure writes eventually become visible. In the absence of updates, write nodes must push the log either with their current clock value or by synchronizing with peer nodes.

Finally, the data tier handles replication between regions and guarantees the durability of the data. The architecture supports partial replication of the data in the data tier. Additionally, it does not assume a specific data model, thus being adaptable to different storage services.

6.1.2 Design Choices

The design of the reference architecture considered several aspects, not only at the architectural level but also concerning the type of clock to use, the way to track the causal dependencies of each write, and the synchronization strategy. Below, we describe the most relevant aspects considered during the architecture's conceptualization and justify our decisions.

6.1.2.1 Clock Type

Based on the literature review, the available clock alternatives were the following: **a)** Physical Clocks, **b)** Lamport Clocks, **c)** Vector Clocks, or **d)** HLCs.

Considering the need to ensure the *NOC properties* of ROTs, option (a) was discarded because clock skew can result in blocking behavior or multiple retries, which disrespects the **N** and **O** *NOC properties*, respectively. Option (c) would result in $O(N)$ sized messages, which is also incompatible with the **C** *NOC property*. So the choice was reduced to options (b) and (d). Both alternatives allow advancing the clock to match the timestamp of incoming events, and version writes using a single scalar timestamp. However, contrary to logical clocks, HLCs provide reference to physical time, making it possible to index versions in a user-friendly way. Moreover, HLCs can advance in the absence of updates. Given their added benefits, the choice fell on HLCs.

6.1.2.2 Dependency Tracking

The choice of using a HLC to timestamp write operations also enables tracking causality through a single scalar timestamp. Other strategies that explicitly track dependencies or where dependency metadata grows linearly with the number of DCs or partitions would possibly be incompatible with the **C** *NOC property*. However, the decision to use a single timestamp to track each version's causal dependencies trades off the metadata overhead for a potential increase in staleness.

6.1.2.3 General Strategy

Considering the review performed in section 3.2.3, the design of the reference architecture could have adopted one of the following strategies to enforce causality: **a)** Dependency Checking, **b)** Sequencer-based, **c)** Stabilization Protocol, **d)** Optimistic approach, or **e)** PORT's approach [58].

However, given the requirement for performance-optimal ROTs, options (a) and (d) were discarded, the first because it is incompatible with constant metadata and the second because it either requires blocking or multiple retries. In the literature, option (c) was generally coupled with a coordinator-based approach, where ROTs required two communication rounds. Option (b) may inhibit horizontal scalability by introducing a single element to sequence writes. In PORT's design, new or inactive clients may see arbitrarily stale data.

All options considered, the proposed architecture combines strategies (c), (d), and (e) with the architectural approaches described in section 3.3. Similarly to sequencer-based approaches, it totally orders writes in write nodes. However, instead of having a single sequencer process per replica, it uses one write node per shard. Like stabilization-based approaches, read nodes determine the stable frontier from where it is safe to read. Finally, like PORT, ROTs take a single round of communication to the server.

6.1.2.4 Clock Synchronization

In a sharded system, shards may receive write requests at different rates. Regardless, in the absence of updates to a shard, the system must not indefinitely delay making the writes from other shards visible. To that end, most systems use heartbeats to communicate the clock value of a shard that has not processed new writes for some time. When using physical clocks, on the other hand, visibility may depend on NTP synchronization. In the proposed architecture, if a write node does not apply a write for Δ time, a synchronization strategy must be used to ensure progress. To that end, a node may synchronize its clock with other write nodes and persist the log with its new clock value. Given the usage of HLCs, another alternative is to use the current clock value, which advances in the absence of events, to timestamp the log. The architecture does not enforce a concrete strategy, leaving this decision to the implementation.

6.1.2.5 Read and Write Nodes

The reference architecture decouples reads and writes by adopting an architecture where write nodes handle the write requests issued for their respective shard and read nodes handle ROTs issued for items available in their region. Regardless of using one write node per shard, this choice results in higher write latency than multi-master approaches. However, assuming partitions are disjoint, it avoids concurrent conflicting writes because each write node can independently queue and process write requests in order of arrival. Additionally, it allows the system to scale independently. In particular, in read-heavy systems, additional read nodes can be allocated to handle increased read load without affecting the ordering of operations.

Furthermore, since read operations do not modify data, read nodes can execute them concurrently without conflicts. Also, as writes are asynchronously persisted, write nodes may require disk access to back up new writes before they are stored in the storage service (e.g., writing in a memory-mapped file before returning the response to the user). By decoupling reads and writes, any contention due to I/O operations does not affect reads. Finally, in two-sided platforms, where content consumers and producers require different functionalities, separating read and write handling enables each component to cater to the specific needs of its audience.

6.1.2.6 Replication

Partial replication provides a way to balance propagation and storage costs. However, upon low data locality in the data access pattern, requests may be forwarded to remote servers, impacting read performance. Furthermore, partial replication makes it more challenging to ensure CC because ROTs may need to read from remote servers whose `stableTime` may differ from the client's local server. To solve this problem, PaRiS [85] UST protocol establishes a `stableTime` across all servers. Even though the reference architecture could support a similar strategy, that would result in two rounds of communication for ROTs — one to get the transaction's snapshot and a second round to the remote DCs. Therefore, the architecture assumes data can be partitioned

to ensure full data locality, implying that client requests are restricted to the partitions of their local server. This choice ensures a lower propagation time of the requests and reduces staleness compared to a global stabilization protocol at the cost of constraining the partitioning strategy. Regardless, read nodes may keep track of the full data set.

6.1.2.7 Separation of Concerns

This cloud-native reference architecture integrates novel ideas from the literature into a layered architecture akin to the one proposed by Bailis *et al.* [8], separating the causality concern from data replication and durability. Consistency guarantees are enabled by the two top layers using a strategy similar to Wren's [84] and PaRiS's [85], where a ROT is assigned with a snapshot that is the union of a causal snapshot installed by every partition in the client's local region and a client-side cache. However, the ordering of operations is handled by the write node of each shard. The underlying data store manages replication and durability. By decoupling these properties, we effectively bring portability to CC, providing an architecture that can upgrade the consistency guarantees of existing cloud storage services, which provide higher data accessibility and relieve organizations from the intricacies of data replication and reliability.

The reference architecture is divided into three tiers, promoting modularity and allowing for better maintainability and technology independence between tiers. The top client-side tier provides an interface for end users or applications to perform the desired requests, abstracting the connection with the compute tier through a client library. The computing tier is responsible for processing client requests, implementing business rules, and persisting the data in the data tier. Finally, the data tier handles data replication and guarantees durability.

6.1.3 Assumptions

The reference architecture's design holds onto the following assumptions (As) regarding the underlying storage service, the distribution of client requests, and the partitioning of the data:

A1. The data store enables managing replication and specifying the partition in which a write or read request must be performed.

A2. The data store ensures eventual convergence.

A3. The data store's interface allows retrieving a data item whose identifier is lexicographically higher than the specified value.

A4. A read node in region R must only respond to read requests for partition P if it stores a replica of P.

A5. A write node W responsible for partition P must only accept write requests for objects stored in partition P.

A6. Data partitions are disjoint.

A7. Clients are sticky to a read node.

A8. Client requests are restricted to the partitions of their local read node.

A9. A client only sends a request when it has received the response for the previous one.

Assumption **A1** is necessary to ensure that write nodes can specify the data partition where they want to persist the log. It also allows read nodes to specify where to read from and may enable more efficient placement of the servers to minimize the latency of client requests and the round-trip time to the storage layer.

Assumption **A2** guarantees that all replicas eventually converge to the same state, which is required to ensure liveness.

Assumption **A3** requires that items can be range filtered by their identifier, which is necessary to ensure read nodes can retrieve monotonically increasing log versions.

Assumption **A4** ensures that read nodes do not violate consistency by answering a request for a partition for which they are not responsible. Similarly, **A5** ensures the correctness of the log by impeding write nodes from registering an update in the log of another shard.

In **A6**, we assume that each partition stores a non-overlapping subset of the data. This assumption is recurrent in the literature because it simplifies replication and consistency. In our case, it also facilitates building a log per shard and avoids the complexity and overhead of log synchronization.

To guarantee that each client session will always see monotonically increasing versions of the data, **A7** requires clients to be sticky to a read node. The reference architecture may be extended to overcome this restriction, but we decided to leave that enhancement for future work section 9.6. In this regard, **A8** restricts client requests to partitions available in their local read node.

Finally, **A9** assumes that each client may only issue a single request at a time, making it simpler to ensure CC across a client session.

6.1.4 Checkpointing

The reference architecture relies on log-sequencing to establish a total order among each shard's versions. Therefore, it was necessary to establish a log pruning strategy to avoid unbounded growth of the log and the resulting storage and network overhead.

In this regard, write nodes periodically replace a prefix of the log with a checkpoint. However, due to the lack of ordering guarantees of the storage layer, write nodes must first determine when it is safe to perform a checkpoint without compromising the consistency guarantees of the reference architecture. Otherwise, as read nodes always fetch the most recent log from the data store, the read node may miss a version pruned by a recent checkpoint.

For example, consider a system configuration where a read node R is responsible for partitions P_1 and P_2, which hold keys x and y, respectively. W_1 and W_2 are the write nodes responsible for partitions P_1 and P_2. x_n represents version n of key x, and t_{x_n} represents the timestamp of version x_n. Now, consider the following scenario:

- A client writes versions x_1, y_1 and x_2 in this order, which get timestamped with t_{x_1}, t_{y_1} and tx_2, where $t_{x_1} < t_{y_1} < t_{x_2}$.
- W_1 pushes its log with versions x_1 and x_2 and timestamps it with t_{x_2}.
- W_2 pushes its log with y_1 and timestamps it with t_{y_1}.
- The client writes version x_3, which gets timestamped with t_{x_3}.
- W_1 replaces the prefix of the log by a checkpoint, keeping only its last version (x_2). It then attaches the newest version x_3 to the log and pushes it with timestamp t_{x_3}.
- R fetches the last log of each partition and gets the logs with t_{x_3} from partition P_1 and t_{y_1} from partition P_2. It computes the `stableTime`, which would be t_{y_1}. Due to the checkpoint performed in W_1, R does not see any version of x stable in t_{y_1}. Therefore, upon a ROT of keys x and y, R would return a null value for x and version y_1 for y, even though x_1 happened-before y_1, violating CC.

To overcome possible anomalies that may result from performing an arbitrary checkpoint, a write node must determine the minimum `stableTime` across the read nodes responsible for his partition. The architecture does not enforce a concrete approach to determining the minimum `stableTime`, leaving this decision to the implementation. However, it requires that write nodes use that value to perform the checkpoint: they must remove all but the latest version that happened before the minimum stable time (i.e., leaving the nearest version of each item whose timestamp either matches the minimum stable time or happened before that timestamp).

6.1.5 Garbage Collection

To prevent memory leaks and keep storage bounded, discarding older versions of the data from compute nodes and deleting old log versions from the data store is also necessary.

To that end, read nodes may garbage collect the versions from each item's version chain whose timestamp is lower than their `stableTime`, provided that the latest stable version of each item is preserved and that no active ROT has a lower snapshot. The checkpointing protocol introduced in the previous section also makes it possible to garbage-collect old data versions at the write node.

Considering the architecture's applicability to cloud-native systems and its support for cloud storage services, which may charge for storage capacity and usage, a garbage collector process may periodically discard old log versions, keeping the number of log versions bounded. However, when combined with the checkpointing strategy described in the previous section, garbage collection at the data store will impact the number of stale versions available and, thus, may inhibit developers from auditing the system. In this sense, it is essential to balance the trade-off between storage cost and loss of historical versions of the data. This detail is further discussed in the validation of our solution section 8.5.

6.1.6 Fault Tolerance

Assuming that failures are fail-stop (i.e., "components halt in response to a failure instead of operating incorrectly or maliciously"[53, p. 409]) and that the underlying storage service follows the assumptions described in section 6.1.3, this section covers the minimal set of failure-scenarios and discusses possible failure-recovery techniques.

Client Failure

Similarly to PaRiS [85] and Wren [84], a client failure does not affect the system or other clients because clients only store metadata regarding their last write operations, which have already been committed to the data store. From the server's perspective, the client stopped issuing requests, so no recovery is necessary. For clients, however, the loss of state may affect the causality of his operations (i.e., CC is only guaranteed within a session).

A possible approach to support client state losses would be to adapt the client recovery protocol of ViewStamped Replication [51] but have the client sequence his requests with a vector clock with one entry per partition.

Another possible issue, however, may arise if the client connects to another read node when it recovers because the read node's `stableTime` may be lower than the last one observed by the client. This scenario and other possible client failure modes are left for future work.

Write Node Failure

In order to ensure durability when a write node fails and recovers, a write must be confirmed to the client only after storing the log on disk (e.g., using a memory-mapped file). Because writes are idempotent, if the write node fails by not responding, the client library may adopt an exponential backoff technique to retry the request.

When a write node fails and recovers, it may recover the log from the disk. Since write nodes handle one request at a time, the last write may have been registered on the log but not confirmed to the client. In that case, the write node may need to store metadata about the client of the last request it was processing to ensure that the client stores the new version in the cache and updates his `lastWriteTimestamp`.

In case of hardware failures, a backup write node may fetch the latest log from storage and advance the clock accordingly. The checkpointing protocol improves the recovery time. The write node may, however, have failed before storing the log in the data store. In that case, some cloud storage services offer robust recovery solutions (e.g., Amazon EBS volumes are automatically replicated to protect against failures and can be attached to other machines). Another alternative would be to use a replication recovery protocol where fail-over nodes handle the same operations.

In order to support additional failure modes, the write node may be adapted to use a robust state machine replication protocol (e.g., [51]). Other failure recovery strategies are discussed in the future work section.

The failure of a write node also blocks the progress of the `stableTime` on the read nodes, but only as long as the node has not recovered or a backup has not taken over.

Read Node Failure

Suppose a read node fails by not responding. In that case, the client may perform an exponential backoff to retry the request. After some retries, the client may fail over to another read node (if available). In this case, the protocol may be extended so that the client library stores the last `stableTime` seen by the client and sends it to the fail-over read node to ensure the client always observes a causally consistent snapshot of the data. If a snapshot is not available, the read node may respond with an error message. While the new read node is not up to date with the previous one, the client library may retry the read operation, avoiding a synchronous read on the read node. When replacing a read node, that node may recover the log from the data store, determine the `stableTime` and proceed like a fail-over node.

Data Store Failure

Suppose a log version fails to be persisted in the data store. In that case, the write node may combine two approaches: retry using an exponential backoff technique and, given that the log is persisted periodically, persist the log in the next run of the Log Pusher.

6.2 Comparative Analysis

The proposed reference architecture integrates ideas from some of the systems reviewed in section 3.2.1, which support ROTs, and from the approaches described in section 3.3, presenting progress towards a general solution for applying CC to real-world read-heavy systems. Table 6.1 extends the table presented in section 3.2.2 with the properties of MongoDB [91] and Bolt-on [8], which, however, do not provide ROTs.

First, like several causally consistent systems reviewed in section 3.2.1, the reference architecture provides ROTs and non-transactional writes.

Similarly to MongoDB's causally consistent implementation, each shard can only be updated by a single node (the write node) and reads must be directed to read nodes. However, while in MongoDB, reads may be performed from the primary, this reference architecture separates read and write operations, making the primary a fixed write-only node.

Another similarity to MongoDB's architecture is its log-based approach and state machine behavior.

The reference architecture uses HLCs to timestamp events and a single timestamp to track dependencies. That is also the case of MongoDB and of PaRiS [85], Wren [84], and Contrarian [23], which rely on stabilization protocols to determine which updates can be made visible. However, PaRiS tolerates intra-DC requests, thus requiring the stable frontier to be computed across all nodes. In contrast, our reference architecture assumes full data locality. Moreover, it avoids the

System	Taxonomy	Clock	Replication	ROT performance optimality		
				Non-blocking	Rounds	Metadata
COPS [53]	ROT	Logical	Full	✓	≤ 2	$O(D)$
Eiger [54]	ROT & WOT	Logical	Full	✓	≤ 3	$O(D)$
ChainReaction [5]	ROT	Logical & Physical	Full	✗	≥ 2	$O(M)$
Orbe [24]	ROT	Logical & Physical	Full	✗	2	$O(1)$
GentleRain [25]	Snapshot & ROT	Physical	Full	✗	$\leq 2 + off\text{-}path$	$O(1)$
SwiftCloud [98]	Generic	Logical	Partial*	✓	0	$O(1)$
Cure [4]	Generic	Physical	Full	✗	$2 + off\text{-}path$	$O(M)$
Occult [65]	Generic	Logical	Full	✓	≥ 1	$O(N)$
COPS-SNOW [56]	ROT	Logical	Full	✓	$1 + off\text{-}path$	$O(D)$
POCC [86]	ROT	Physical	Full	✗	2	$O(M)$
Wren [84]	Generic	Hybrid	Full	✓	2	$O(1)$
Contrarian [23]	ROT	Hybrid	Full	✓	2	$O(M)$
PaRiS [85]	Generic	Hybrid	Partial	✓	2	$O(1)$
Eiger-PORT [58]	ROT & WOT	Logical	Full	✓	1	$O(1)$
MongoDB [91]	Reads & Writes	Hybrid	Full	-	-	-
Bolt-on [8]	Reads & Writes	Logical	Full	-	-	-
Reference Architecture	ROT	Hybrid	Partial	✓	1	$O(1)$

Table 6.1: Characterization of geo-replicated causally consistent systems. Partial* stands for partial replication at the client. D stands for the number of dependencies, M is the number of DCs and N is the number of partitions. MongoDB [91] and Bolt-on [8] do not support ROTs; thus, they are not classified according to the last columns. The last row classifies the proposed reference architecture (Adapted from [58], [85] and [23]).

extra communication round required by coordinator-based approaches by partitioning read nodes by region instead of shard.

Like PaRiS and Wren's transactional protocol, our design requires ROTs to be assigned with a snapshot that is the union of a causal snapshot installed by every shard in the client's local region and a client-side cache.

For clock synchronization and liveness, in PaRiS and Wren, servers recur to heartbeats, and MongoDB generates a no-op to advance the clock on the primary. In this reference architecture, nodes may persist the log with their current clock or synchronize with peer nodes. Furthermore, like in PaRiS, where the client piggybacks the timestamp of his last write transaction *hwc*, and in MongoDB, where clients attach the `operationTime` to subsequent operations, write nodes may advance their clock to match the `lastWriteTimestamp` of the client.

Like SwiftCloud [98], the reference architecture follows a log-based approach. However, while SwiftCloud requires servers to keep a full copy of the data store and relies on CRDTs for convergence, our proposed architecture enables read nodes to maintain a partial copy of the data store. It avoids the complexity of log merging by adopting a log per shard handled by a dedicated write node.

Like Bolt-on's approach, the reference architecture enhances modularity and supports existing cloud storage services, bringing portability to CC. However, given our solution's visibility and read performance requirements, we integrate existing stabilization and sequencer-based approaches where the version's timestamp identifies its causal dependencies instead of requiring that each

version explicitly tracks individual dependencies. Like Bolt-on's optimistic algorithm, where a resolver process asynchronously updates the local store, the reference architecture uses a Log Puller to fetch the log asynchronously.

In Bolt-on's work, the authors mentioned the possibility of avoiding overwrites by storing a new data item for every write [8, p. 17] but followed a different approach due to the storage costs of this alternative. In contrast, our reference architecture stores a new item for each write. Still, it gets around this limitation using a log-based approach where multiple writes are batched in a single update and through checkpointing and garbage collection, avoiding unbounded storage and network overhead.

Finally, our reference architecture targets read-heavy systems and, thus, was designed to enhance read performance. In this regard, like SwiftCloud and Eiger-PORT [58], the reference architecture enables performance-optimal ROTs. However, it ensures these properties through different design choices, which will be further analyzed in section 8.3.

6.3 Summary

In this chapter, we have discussed the core decisions and reasoning behind the proposed reference architecture and compared it with the systems and architectures identified in the literature review, addressing **RQ4**.

Section 6.1 overviewed the main components and features of the reference architecture design. Firstly, the architecture is organized into three tiers: the client tier, where clients use the client-side library to issue ROTs and write operations; the computing tier, consisting of server nodes specialized in either read or write operations; and the data tier, which provides data durability and handles replication. To timestamp events, write nodes use HLCs. The architecture combines several strategies described in section 3.2.3. For instance, like sequencer-based approaches, write nodes order the writes in a log. Similarly to MongoDB [91], each write node handles the writes of a single shard. Like stabilization-based approaches, read nodes also determine the stable frontier from where it is safe to read. Moreover, like Bolt-on [8], the architecture is storage-agnostic and capable of upgrading the consistency of existing cloud storage services.

Furthermore, this section has described the assumptions on which the architecture is built and outlined its checkpointing and garbage collection strategy. Finally, it briefly touched upon possible failure recovery strategies.

Section 6.2 compared the reference architecture with the systems and approaches surveyed in the literature review. First, regarding the clock, the reference architecture shares similarities with PaRiS [85], Wren [84], Contrarian [23], and MongoDB [91], using an HLC to order operations. It supports partial replication on the server side but limits clients to the partitions of their region. In contrast, PaRiS [85] enables clients to access data in partitions of other DCs, and SwiftCloud [98] only supports client-side replication. Like Eiger-PORT [58] and SwiftCloud [98], the proposed architecture enables the performance-optimal properties of ROTs. Finally, by integrating the ideas

from Bolt-on's [8] approach, the reference architecture enhances modularity and supports existing cloud storage services, bringing portability to CC.

Chapter 7

Reference Architecture Realization

The previous chapter justified the design choices of the proposed reference architecture. This chapter discusses the implementation of a prototype that realizes the reference architecture using Amazon S3 as the underlying storage service, presenting progress toward **RQ5**. Additionally, it discusses how this implementation enables *value semantics*, touching upon **RQ6**. The source code of this prototype implementation and its instructions are available in this work's replication package [30].

Section 7.1 starts by describing the technology, tools, and cloud services used to develop the prototype system. Then, section 7.2 presents an overview of the system's components. Section 7.3 describes the system's programming interface and is followed by an overview of the state of each system component (section 7.4) and by a description of the protocols used for ROTs and write operations (section 7.5). Section 7.6 explains the implementation of the HLC and section 7.7 describes the asynchronous log propagation and the clock synchronization algorithm. Section 7.8 describes the checkpointing and pruning strategy and the garbage collection of stale versions. Section 7.9 discusses how the prototype system enables *value semantics*. Finally, section 7.10 summarizes the topics above.

7.1 Technology Stack and Cloud Infrastructure

The prototype was developed in Java, using Maven as the build tool. It uses the AWS's Java SDK to interact with Amazon S3 and Amazon Elastic Compute Cloud (Amazon EC2). Amazon S3 was used as the underlying storage service. It is an object storage service with high scalability and durability that delivers read-after-write consistency. Amazon EC2 was used to deploy the system components, namely the read and write nodes and the clients. It is a cloud computing service that provides many instance types optimized for different use cases.

For client-server communication, the prototype uses Google Remote Procedure Call (gRPC), a high-performance, open-source gRPC framework with the default Protocol Buffers for serializing the data. For the logs, we used JSON format.

Additionally, Localstack was required during the development of the prototype to emulate the AWS cloud services in a local environment.

Lastly, to containerize the prototype system and its dependencies and make it easier to test and deploy, we built a Docker image that can be used to run any of the system components.

7.2 System Overview

This prototype implements a causally consistent multi-versioned key-value data store, using Amazon S3 as the underlying storage service. It allows clients to create or update a single key or to extract a consistent view of a set of keys. Additionally, it supports atomic write operations, which enable the client to conditionally set the value of a given key.

It implements two gRPC server processes and a client process destined to be deployed in the physical entities identified in the reference architecture, namely in write and read nodes and in each client machine. Figure 7.1 illustrates the system's architecture and core components.

Client Process. The client process works as a client-side library and can be used by client-side applications to perform ROTs and write operations. The client process is single-threaded, and forwards client read requests to its designated read node and write requests to the respective write nodes through gRPC stubs. The client process also keeps track of the `lastWriteTimestamp` and stores unstable writes in a private cache.

Write Nodes. Write processes are single-threaded gRPC servers that handle write requests to a data partition in order of arrival. They use a HLC to timestamp each write and store them in their storage (`WriterStorage`). Periodically, a thread (`StoragePusher`) builds the log and persists it in the Amazon S3 bucket of the respective data partition. Each Amazon S3 bucket stores the logs of a single partition, and write nodes only persist the log in one bucket. Replication to other buckets is specified in the bucket's configuration and handled by Amazon S3.

Read Nodes. Read processes are multi-threaded gRPC servers that concurrently process ROT requests. They asynchronously fetch the most recent logs of a predetermined set of partitions from the respective Amazon S3 buckets, parse them, and update their storage (`ReaderStorage`). These tasks are periodically performed by the `StoragePuller`. Read nodes keep track of the `stableTime`, a timestamp below which all versions are stable in that node. Versions with a timestamp above the `stableTime` cannot be visible to clients.

7.3 Programming Interface

The client library provides the following operations to the client:

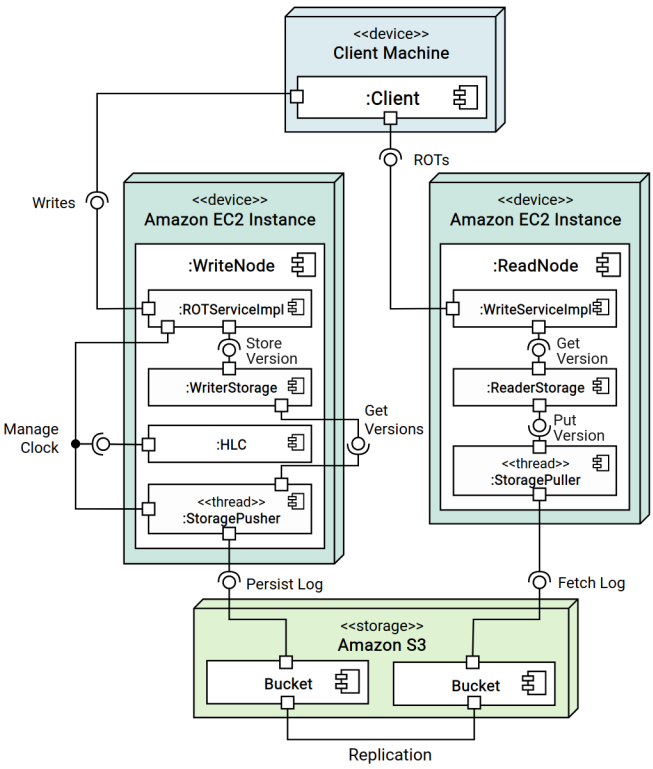


Figure 7.1: Prototype system components UML deployment/component diagram. The system realizes the reference architecture using Amazon S3 as the storage service and Amazon EC2 to deploy the servers (i.e., write and read nodes). The `Client` process works like a client-side library, handling the interaction with the servers and keeping metadata of his previous writes. The `WriteNode` server exposes a gRPC service that provides write operations. Write nodes use a `HLC` (i.e., the HLC) to order writes and use a (`WriterStorage`) for storing the new versions. A `StoragePusher` thread reads the versions from storage, builds the log, and persists it in the Amazon S3 bucket of its partition. Amazon S3 handles replication to other buckets. The `ReadNode` server uses the `StoragePuller` thread to fetch the logs of the partitions it stores from the appropriate buckets. It updates its storage (`ReaderStorage`) and the metadata required to ensure CC. Read nodes expose a gRPC service that accepts ROTs.

- $\langle V, stableTime \rangle \leftarrow$ `requestROT(`$\langle keys \rangle$`)`: Reads the set of keys specified in the input parameter and returns the set of values (V) that correspond to the state of the requested keys at `stableTime`.
- $writeTimestamp \leftarrow$ `requestWrite(`$key, value$`)`: Creates a new version of the item identified by key with value $value$ and returns the timestamp assigned to that version.
- $\langle writeTimestamp, currentVersion \rangle \leftarrow$ `requestCompareVersionAndWrite(`$key, value, expectedVersion, expectedValue$`)`: If the current version of the item identified by key has the timestamp $expectedVersion$ and the value $expectedValue$, it creates a new version of that key with value $value$ and returns the $writeTimestamp$ assigned to the new write. Otherwise,

it returns the last timestamp of that key (*currentVersion*). If *expectedValue* is not defined, the write is performed if the current version is the one specified in *expectedVersion*.

- $\langle writeTimestamp, currentVersion \rangle \leftarrow$ `requestCompareVersionAndWrite` $(key, value, expectedValue)$: If the current version of the item identified by *key* has the expected value (*expectedValue*), it creates a new version of that key with value *value* and returns the *writeTimestamp* assigned to the new write. Otherwise, it returns the last timestamp of that key (*currentVersion*).

7.4 State

As described in chapter 6, the client library and server nodes must keep metadata to guarantee CC.

Client State. The client process uses a map data structure that behaves like a private cache, keeping track of the unstable versions written by the client. For each key, the map stores the value and timestamp of the last unstable version of that key. Additionally, to guarantee monotonic writes, the client keeps the timestamp of his last write (`lastWriteTimestamp`).

Write Node State. Write nodes store the version chain of each key in a dedicated `ConcurrentSkipListMap`, a thread-safe sorted map that allows efficient access to the key's versions by timestamp and provides the ability to get an immutable snapshot of the data in a lock-free way. This data structure is beneficial for atomic write operations, where its `lastEntry` method provides access to the last version of a given key. The `keyVersions` `ConcurrentMap` tracks the version chain of each key. Each write node also has access to a HLC, which assigns timestamps to new item versions. The clock implementation is further detailed in section 7.6.

Read Node State. Like write nodes, read nodes store the version chain of each key in a dedicated `ConcurrentSkipListMap`. This data structure provides a `floorEntry` method that returns the highest entry lower or equal to a key, enabling read nodes to access the last stable version of a key when computing the response to a ROT. Additionally, to guarantee that read nodes see monotonically increasing versions of each partition's log, they also keep track of the maximum log timestamp they have seen from each partition (`partitionsMaxTimestamp`) and use that value to compute the `stableTime` — the maximum log timestamp that has been seen from all partitions.

7.5 Protocols

This section describes the protocols for executing each of the operations specified above.

ROT: A client performs a `requestROT` for a set of keys through its client-side library, which forwards the request to the server through a gRPC message, whose format is depicted in listing B.1. The read node uses the current `stableTime` as the transaction's timestamp, ensuring all keys will be read from the same causal snapshot. For each key, the read node retrieves the last stable version (i.e., the highest version that is at most equal to the transaction's timestamp) using the `floorEntry` method described before. The response to the client includes the value of all requested keys and the transaction's timestamp. The client transverses its cache and removes all versions whose timestamp is lower than the `stableTime`. Then, for each key requested, it verifies if a version remains in the cache. If so, it replaces that version in the response set. Finally, the updated set of keys is returned to the client. Figure 7.2 illustrates the flow of a ROT.

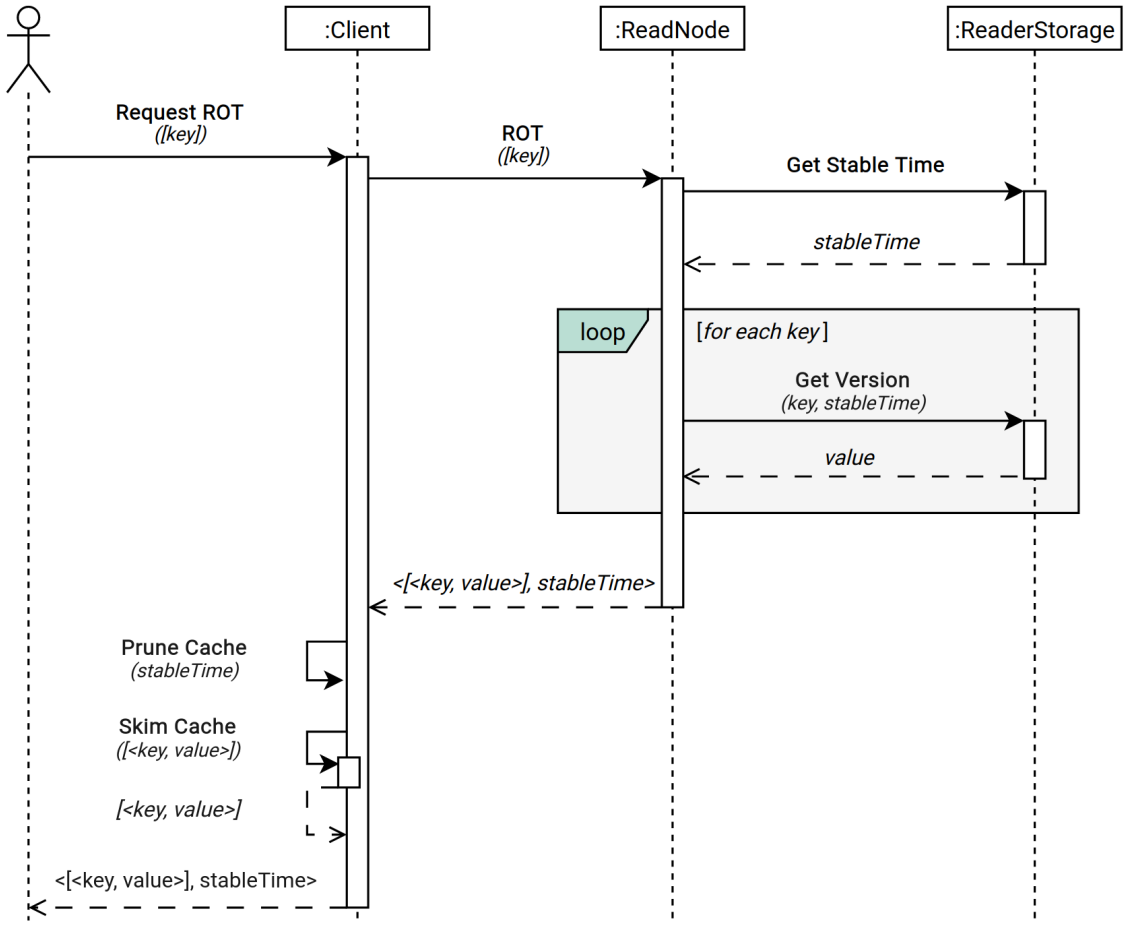


Figure 7.2: ROT request UML sequence diagram. Read nodes use the `stableTime` to get the last stable version of each key from their storage and return the response to the client. The client-library prunes stable versions from its cache using the `stableTime` returned in the response and updates the versions to be returned to the client if it finds a more recent version in the cache.

Writes: A client uses the `requestWrite` method to write a value v to key k. The client library builds the respective gRPC message, attaching the `lastWriteTimestamp` of the client, and

redirects it to the write node that is responsible for k's partition. The message format used for write requests is depicted in listing B.2. The write node uses its HLC to assign a timestamp to the new version of k and assigns value v to that version, stores the new version in k's version chain, and returns the version's timestamp to the client, which updates his `lastWriteTimestamp`. Figure 7.2 depicts the flow of a write request.

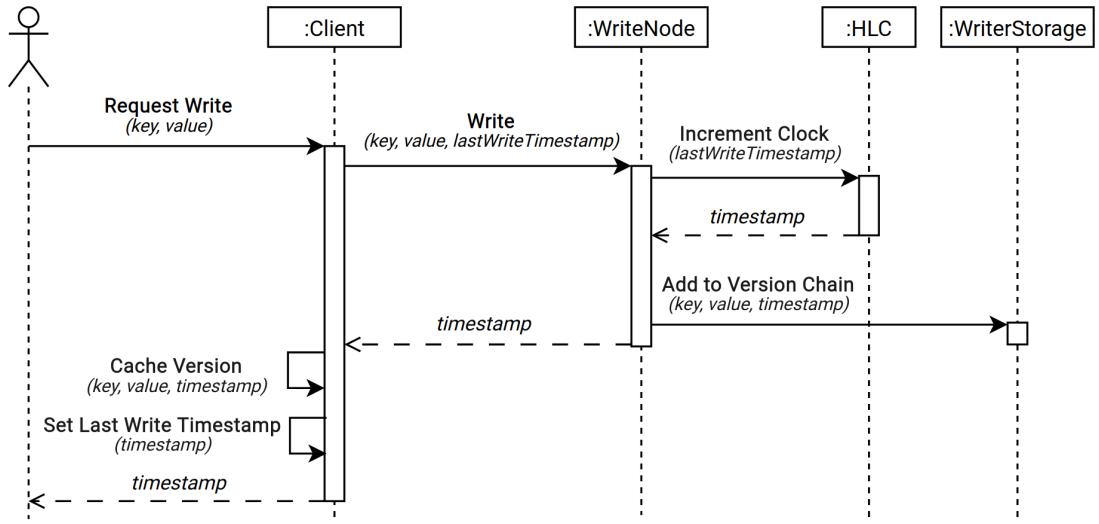


Figure 7.3: Write request UML sequence diagram. The client requests to read a set of keys. The client library appends the timestamp of the client's last write to the request. The write node increments its clock (HLC), getting the timestamp of the new write, adds the new version to the key's version chain, and returns the response to the client library, which updates its cache and the timestamp of the client's last write.

Atomic writes: When a client issues an atomic write, the client library converts the request to a gRPC message that, like write operations, includes the `lastWriteTimestamp` of the client but also two optional fields: `expectedVersion` and `expectedValue`. The format of the messages is specified in listing B.2. If at least one of these fields is set, the write node will first retrieve the current version of the specified key using the `lastEntry` method and compare its timestamp and value with the ones provided by the client. If they match, it can apply the write because it processes a write at a time, and no other process handles writes to this key. The rest of the protocol resembles the one used for writes, returning the timestamp of the new version to the client library. If the atomic write fails, the write node returns the timestamp of the current version of the key to the client library, which may either retry the operation or wait until the read node reflects the current version. The flow of atomic write operations is illustrated in fig. 7.4.

7.6 HLC Implementation

The prototype features an implementation of a HLC using Java atomic variables. This implementation targets two use cases: enabling write nodes to increment the clock upon receiving a new

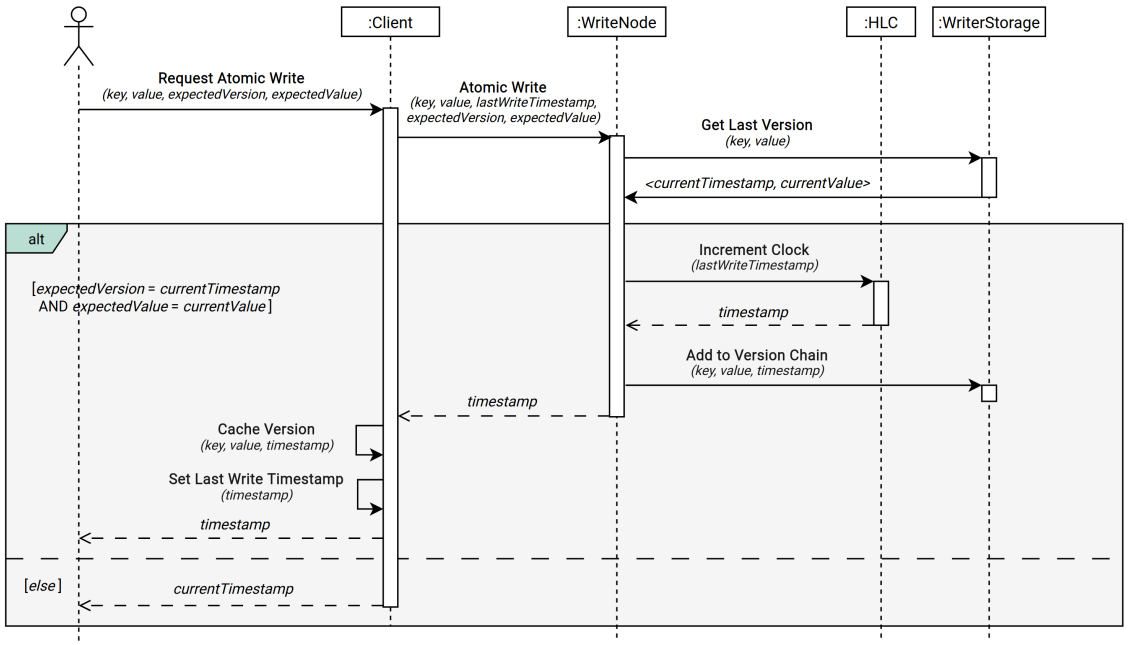


Figure 7.4: Atomic write request UML sequence diagram. The client issues an atomic write to update a key only if its current timestamp is *expectedVersion* and its value is *expectedValue*. The client library appends the timestamp of the client's last write to the client's request. The write is performed if the timestamp and value of the current key version match the ones specified in the request. Otherwise, the request is aborted, and the timestamp of the current version is sent back to the client.

write and synchronizing the clock with other partitions' clocks in the absence of updates. The base clock implementation was later expanded to support additional state tracking, but this section focuses on the base clock implementation.

Each write node uses a `HLC` instance, which tracks the current clock state (`HLCState`) through an `AtomicReference` and provides methods to drive clock state changes. Each method updates the clock state through the built-in `accumulateAndGet` function, which atomically applies a given function to the current value and returns the updated value.

A `HLCState` represents a hybrid timestamp, with its two components: l and c. Just like in the original algorithm [44], the first part (l) maintains the maximum physical timestamp that has been seen so far, and c works as a logical counter to capture the causality when l is equal in two timestamps.

Both parts are encoded as primitive longs and initialized to zero. To generate the timestamp that corresponds to a `HLCState`, each part is padded with leading zeros and encoded as a string with a fixed size (twenty characters). Both parts are then concatenated with a "-". This encoding enables timestamps to be lexicographically comparable, which, as explained in the next section, was required due to the selected storage service.

In a HLC, the first component advances with physical time. However, to avoid the overhead of making a system call per write to get the current time, we use a `TimeProvider` thread

that periodically updates an internal time field with the value of the current time in milliseconds (`currentTimeMillis`). This time provider is used upon a clock operation to get the current time.

Considering the use cases previously outlined, the first requirement of the HLC implementation was to increment the clock upon the reception of an incoming event, in this case, a write request. Clients attach the timestamp of their last write (`lastWriteTimestamp`) in subsequent requests, and the timestamp assigned to the new write must be higher than this value. In this scenario, the prototype implements the original algorithm for the reception of a message. Algorithm 1 presents the pseudo-code of this algorithm.

Algorithm 1 Clock algorithm — Receive write request. pt is the physical time, l is the maximum pt value learned so far, and c is a logical counter. l_{prev} is the l before the request, and l_{recv} is the l value of the client's last write, which is attached to the write request (based on [44]).

$l \leftarrow \max(l_{prev}, l_{recv}, pt)$
$c \leftarrow 0$
if $l = l_{prev} = l_{recv}$ **then**
    $c \leftarrow \max(c_{prev}, c_{recv}) + 1$
else if $l = l_{prev}$ **then**
    $c \leftarrow c_{prev} + 1$
else if $l = l_{recv}$ **then**
    $c \leftarrow c_{recv} + 1$
end if
return $format(l, "-", c)$

In the absence of updates, the second requirement was to enable write nodes to synchronize their clocks with other nodes. To that end, nodes must be able to advance their clocks to match the timestamp of other partitions. In this regard, the implementation provides a method that adapts the original algorithm, advancing the clock without incrementing it. Furthermore, unlike the previous algorithm, the physical time is not used to calculate the l component because, if all partitions stopped receiving write requests for some time and kept synchronizing with other clocks, the clock skew would lead to a potentially infinite number of unnecessary synchronization rounds. Algorithm 2 illustrates this algorithm.

7.7 Log Propagation

In order to guarantee that new versions of the data eventually become visible to clients, write nodes periodically persist the log in the Amazon S3 bucket of their partition. Read nodes asynchronously fetch the latest logs from each of their partitions from the respective Amazon S3 buckets and update their copy of the data. An example of the log format is illustrated in listing B.4.

This section explains the behavior of the `StoragePusher` and `StoragePuller` threads, which map to the Log Pusher and Log Puller components of the reference architecture, persisting and fetching the log from Amazon S3.

Algorithm 2 Clock algorithm — Synchronization. pt is the physical time, l is the maximum pt value learned so far, and c is a logical counter. l_{prev} is the l before the request and l_{recv} is the l value of the client's last write, which is attached to the write request (based on [44]).

$l \leftarrow \max(l_{prev}, l_{recv})$
$c \leftarrow 0$
if $l = l_{prev} = l_{recv}$ **then**
    $c \leftarrow \max(c_{prev}, c_{recv})$
else if $l = l_{prev}$ **then**
    $c \leftarrow c_{prev}$
else if $l = l_{recv}$ **then**
    $c \leftarrow c_{recv}$
end if
return l, c

7.7.1 Persisting the Log

Periodically, write nodes update the operation log and persist it to Amazon S3 through a dedicated thread (`StoragePusher`), whose behavior is illustrated in fig. 7.5. On every execution, this thread verifies if new updates have been made since the last push. If new writes have occurred, it uses the timestamp of the last completed write (i.e., the last write that was appended to its version chain) to timestamp the log. Otherwise, if no new updates have occurred since the last push and there is currently no write in progress, this thread synchronizes the clock with the clock values of other partitions. Each write node persists its last clock value in a shared Amazon S3 bucket so that a node whose clock is behind and did not receive new write requests can advance its clock to match others. The `StoragePusher` retrieves from the storage the item versions that are ready to push, builds the log, and persists it to Amazon S3, creating a new Amazon S3 object where the key is the log's timestamp, and the body is the JSON log encoded as a string. For optimization, the JSON log is saved between executions and extended with the new versions before each push, avoiding the overhead of building the log from scratch every time.

In order to implement this strategy for log persistence and clock synchronization, we extended the `HLCState` class with two new attributes (`lastWrite` and `writeInProgress`). The `lastWrite` field tracks the timestamp of the last successful write (i.e., which was already inserted in its version chain). The `writeInProgress` field indicates if a write is being processed. Even though these fields are not clock components, their state determines if the clock must be synchronized and, thus, is directly related to the current clock value.

As we can see in fig. 7.6, the write node starts in its initial state, where no new versions are ready to persist, and no write is in progress. When a new write occurs, the clock is incremented, and its value is used to timestamp the new version. The `writeInProgress` flag is set to `true`. When this write completes, the `lastWrite` is set to the timestamp of the new version so that the `StoragePusher` knows the highest timestamp that can be included in the log. The `writeInProgress` flag is reset. If new writes are received at this stage, then the write node repeats this procedure. When the `StoragePusher` executes, it uses the `lastWrite` field to get

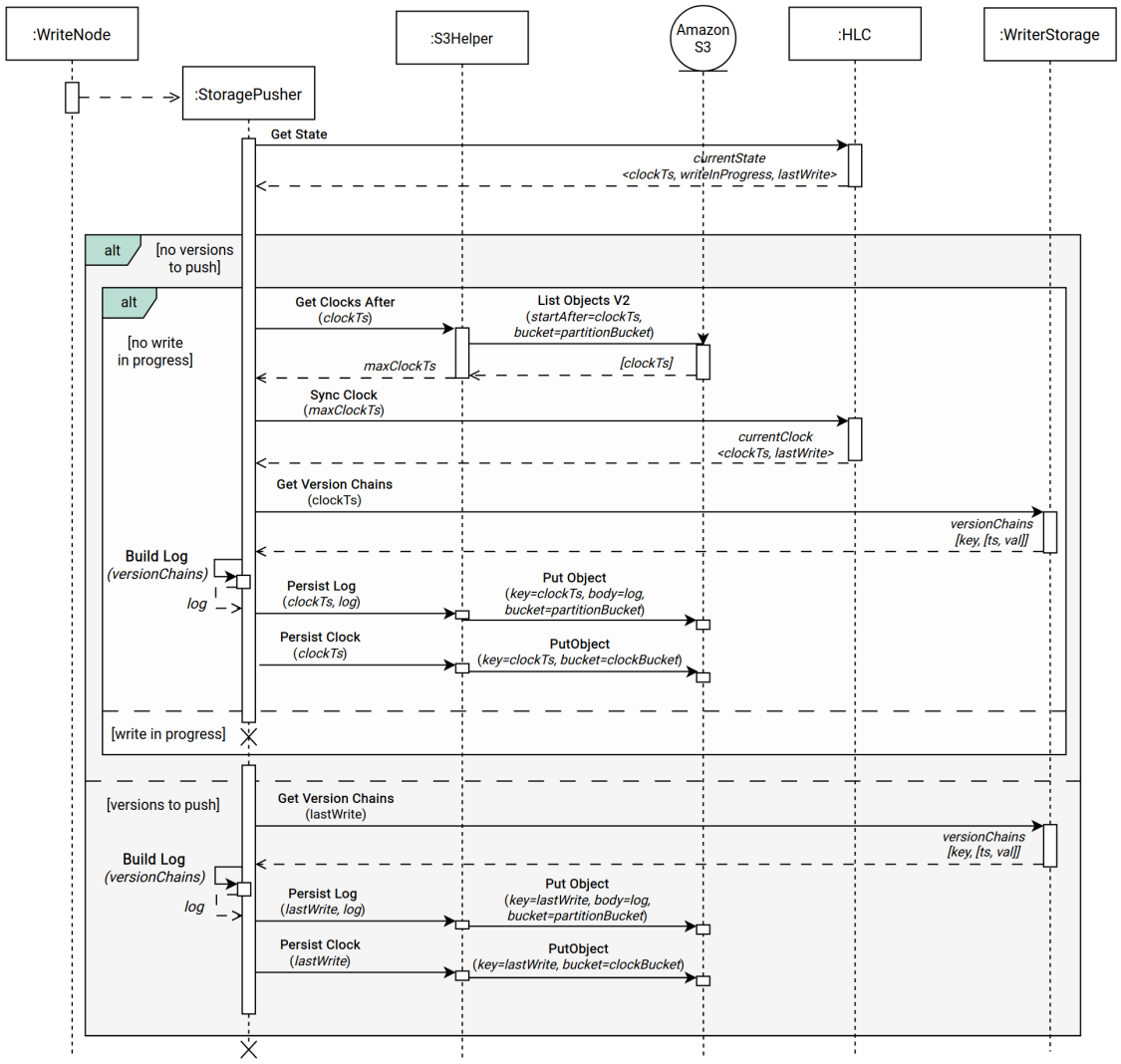


Figure 7.5: Persisting the log UML sequence diagram. Periodically, a `StoragePusher` thread persists the log into Amazon S3 if there are new versions to be pushed and runs the clock synchronization protocol in the absence of updates. First, the thread gets the current clock value of the HLC and the timestamp of the last completed write. The thread is canceled if no versions are ready to persist and a write is currently in progress. If no writes are in progress or ready to be pushed, it tries to run the clock synchronization protocol: it retrieves the last clock values from other partitions from Amazon S3, uses the maximum of those values to advance the clock, and, in case no new writes occurred in the meantime, it uses the new clock value to build the log and persists both the log and the clock value to Amazon S3. If new writes have occurred, then the timestamp of the last write that has completed (`lastWrite`) is used to get the version chains from the storage (`WriterStorage`), version the log and then the log is persisted to Amazon S3

.

the versions up until that timestamp, builds the log, and persists it. If, during the process, no new writes were executed (i.e., the `lastWrite` field did not change), then it resets the field so that in a future execution, the `StoragePusher` knows that no new writes have occurred since the last push. When no writes occur between the executions of the `StoragePusher` thread, the clock is

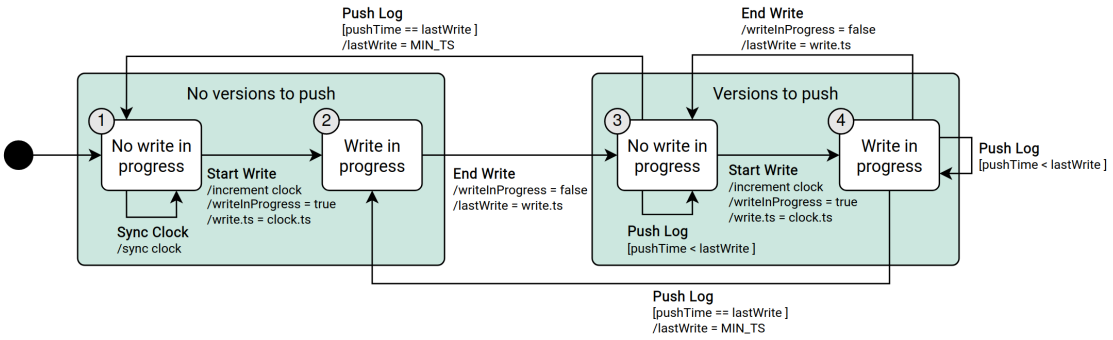


Figure 7.6: Write node state UML state diagram. The write node starts in **State 1**, where no new versions are ready to persist, and no write is in progress. When a new write occurs, the write node increments the clock, using its value to set the timestamp of the new version. The `writeInProgress` flag is set to `true`, and the node moves to **State 2**. When this write completes, the `lastWrite` is set to the timestamp of the new version, and the `writeInProgress` flag is reset. The node moves to **State 3**. If new writes are received, then the write node repeats this procedure, now moving between **State 3** and **State 4**. When the `StoragePusher` executes, it uses the current `lastWrite` to get the versions up until that timestamp. After persisting the log, if no new writes were applied (i.e., the `lastWrite` has not changed), then the `lastWrite` is reset, and the write node moves to **State 1** or **State 2**, depending if a write is in progress. If the write node does not process any write requests for some time, the node remains in **State 1**, and the clock is synchronized with other nodes and used to persist the log.

synchronized with peer nodes.

7.7.2 Fetching the Log

Read nodes periodically fetch the log from the Amazon S3 buckets of each partition they track. However, given that Amazon S3 does not guarantee that logs will be replicated in order, we must adopt a strategy that ensures read nodes see monotonically increasing versions of each partition's log. To that end, the log fetching strategy recurs to AWS's `ListObjectsV2` action, which accepts a `startAfter` field that specifies that only keys that are lexicographically after the provided key should be retrieved. Given that each log version is identified by a timestamp, the timestamps had to be specially encoded to preserve the desired ordering, as described in the previous section.

Read nodes keep track of the maximum log timestamp they have seen from each partition and use that value to perform the subsequent fetch. Upon processing the logs from all partitions, read nodes compute the `stableTime` as the maximum log timestamp seen from all partitions. Figure 7.7 illustrates the behavior of the `StoragePuller` and its interaction with Amazon S3 and with other system components.

7.8 Checkpointing and Garbage Collection

As described in section 6.1.4, the log must be periodically pruned to avoid unbounded storage without compromising the correctness of the system. For that, write nodes must determine the

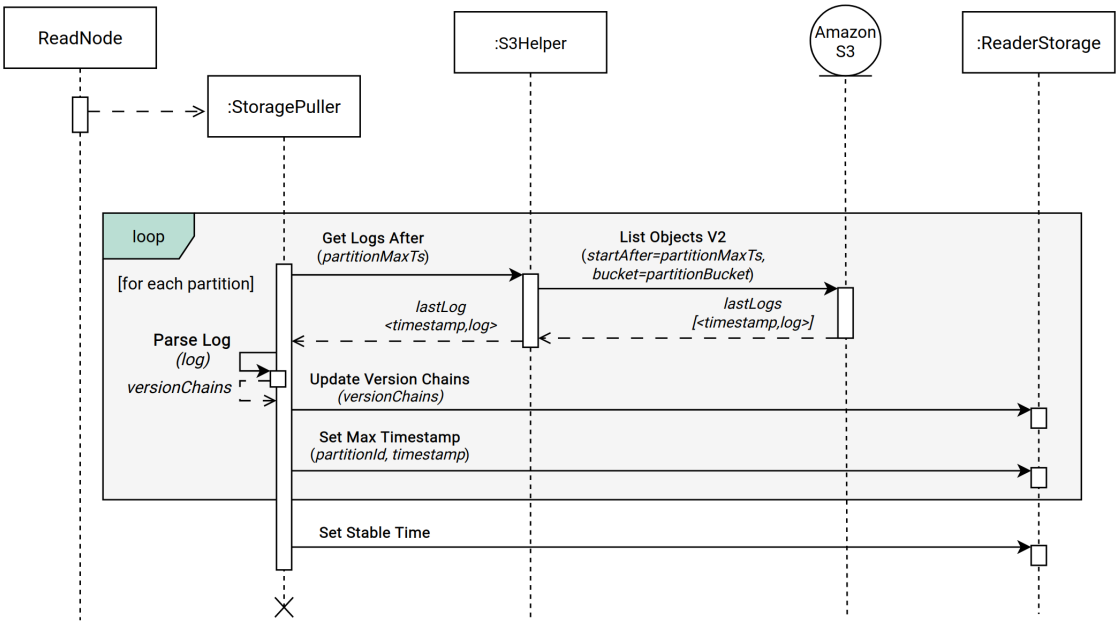


Figure 7.7: Fetching the log UML sequence diagram. The `StoragePuller` thread periodically fetches the last log of each partition from Amazon S3. It parses the log, updates its storage (`ReaderStorage`) with the new versions, and tracks the last log timestamp it has seen from that partition. After processing the logs of all partitions, it sets the `stableTime` as the minimum of the last timestamps seen from all partitions.

minimum `stableTime` across the read nodes that hold a copy of their data partition.

To compute the minimum `stableTime`, read nodes provide an additional gRPC service, depicted in listing B.3, which allows write nodes to request the current `stableTime`. Every *N* run, the `StoragePusher` requests the `stableTime` of each read node and computes the minimum of those times. It then uses this value to prune the log and to garbage-collect old data versions. Once again, the `ConcurrentSkipList` data structure used for storing each version chain proved helpful for determining the versions that can be safely removed. More specifically, the `floorKey` method was used to determine the last stable version of each key (stable frontier), and the `headMap` method was used to get the versions before that stable frontier, which can be safely discarded. Listings B.4 and B.5 exemplify the log's state before and after performing a checkpoint.

Read nodes garbage-collect stale versions using their `stableTime` but only if there are no active ROTs. Otherwise, an active ROT with a smaller timestamp than the current `stableTime` might lose access to the stale versions that belong to its snapshot.

Due to time constraints, we did not implement garbage collection of the logs in Amazon S3, deferring this aspect for future work.

7.9 Value Semantics

In this work, we defined *value semantics* as an interface for computation that enables the perception of *values* in time (*cf.* section 2.3). Considering this definition, our system enables value semantics through four core properties:

Deterministic State Machine Behavior

In our implementation, server nodes assume a state machine behavior. Each server starts at an initial state, where no writes have been applied, and performs the same sequence of deterministic operations to a set of items, reaching the same state. A set of states describes the flow of writes applied to the data items, making it easier to reason about the system's behavior and enabling value semantics.

To guarantee that all replicas execute operations in the same order despite conflicting client requests and thus avoid a non-deterministic system behavior, a write node per data partition sequences the writes to items of that partition in order of arrival. Upon a write request, the write node advances its HLC and uses its value to version the operation, ensuring that applying the operation moves the state forward in logical time. The new version is stored in the item's version chain, a `ConcurrentSkipListMap` that maps each version to its value. The version chains are then encoded in a log and replicated to other replicas (the read nodes). Each replica processes the version chains in timestamp order, only applying them when they are stable. Thus, all read nodes that replicate the same partitions will reflect the same sequence of versions determined by the write node.

Log Versioning

The prototype system adopts a log-based approach where not only are writes versioned, but the log itself is also versioned with the timestamp of the last operation or, in the absence of updates, with the current clock value. Each log represents a snapshot of the partition's state at a given point in time. A sequence of logs portrays the evolution of the partition's state in time, thus enabling value semantics.

Additionally, by using a HLC to assign the timestamps and recording the logs of each partition in a dedicated Amazon S3 bucket, it is possible to use the physical time to index these state versions, which may be of practical use for manual or automatic auditing, or even for end users, as it enables system versions to be indexed using physical time.

Stable View of the Data

In order to have value semantics, the system's state must be stable and perceivable at a given point in time. To that end, the prototype exposes ROT operations to end users, which deliver a snapshot of the system that is stable but possibly stale. As previously described in section 7.4, the prototype leverages the `floorEntry` method provided by the `ConcurrentSkipListMap` data structure

to retrieve the highest stable version of each requested item. In order to support stale reads, just like in other multi-versioned systems, old versions of the data must be stored until a new version becomes stable. As the `stableTime` progresses monotonically and tracks a stable version of the data, the users will always see increasing versions of the system's state.

Atomic Operations

By taking advantage of the immutability of each item version, the prototype extends the programming interface defined by the reference architecture with atomic write operations, where the client can conditionally set the value of an item.

Atomic operations enable a key's state to transparently transition from one value to another and therefore are essential to have value semantics, where values causally succeed like atomic states, each resulting from applying an operation to the other.

The atomicity of these write operations enables total transparency over the system's behavior and determinism. If the current version corresponds to the one specified by the user, then the system state transitions to the new version. Otherwise, the new write is not applied, and the current version is returned to the user or application, which may decide how to proceed.

As write nodes are single-threaded, enabling atomic writes requires the last version of the item to be retrieved with the `lastEntry` method of `ConcurrentSkipListMap` data structure and compared against the value or timestamp provided by the user. If the write cannot be applied, the value of the current version may not be returned to the end user because that might violate causality, given that the write node may be ahead of the client's local read node. Instead, we return the timestamp of the current version so the application may decide how to proceed.

7.10 Summary

In this chapter, we have discussed the implementation of a prototype system that realizes the proposed reference architecture, upgrading Amazon S3 with CC and enabling ROTs and atomic writes. This reference implementation and its execution instructions are available in this work's replication package [30].

First, in section 7.1, we specified the technologies and cloud services used in this implementation. The prototype is developed in Java and uses AWS's Java SDK to interact with Amazon S3 and Amazon EC2. The client-server communication uses gRPC.

Section 7.2 overviewed the system's main components. The client process works as a client-side library, abstracting the communication with the server nodes. Write nodes use a HLC to order writes and periodically persist the log of their partition to the respective Amazon S3 bucket. Amazon S3 handles replication. Read nodes periodically get the logs, update their storage and compute the `stableTime`, from which ROTs can be performed.

In section 7.3, we described the programming interface of the system. The client library exposes ROTs and write operations but also incorporates atomic writes, which enable the client to set the value of a key conditionally.

Section 7.4 detailed the metadata kept by each component and justified the usage of some data structures that enable easy access to a key by timestamp.

Section 7.5 followed up on the previous section by describing the flow of each operation. Upon a ROT, the read node uses its `stableTime` to get the last stable versions of the requested keys and returns the response to the client library, which prunes the cache and updates the versions to be returned to the client, ensuring the client reads his last writes. Upon a write request, the client library attaches the client's `lastWriteTimestamp` to the request, sends it to the write node, which timestamps it with its clock value, adds it to the storage, and returns the timestamp to the client. Atomic writes behave like simple writes but only succeed if the last version of the key is the one specified by the client.

In section 7.6, we explained the base implementation of the HLC. For incoming writes, the clock behaves like in the original algorithm, advancing to be at least as high as the client's `lastWriteTimestamp`. For synchronization with the clock values of other partitions, the algorithm advances the clock to match higher incoming timestamps.

Then section 7.7 explained how the log is persisted and pulled from Amazon S3. Write nodes persist the log using the timestamp of the last completed write and version it with this same timestamp. In the absence of updates, they synchronize with other clocks and persist the log with their current clock value. Read nodes fetch the logs of their respective partitions, ensuring they always get monotonically increasing log versions of each partition. They update their storage and compute the `stableTime` based on the timestamp of the last logs.

Section 7.8 described the checkpointing and garbage collection implementation. Write nodes request the `stableTime` of the read nodes that store their partition and compute the minimum, with which they prune their storage and log. Read nodes use the `stableTime` to garbage-collect stale versions. Garbage collection of the logs in Amazon S3 was not implemented due to time constraints.

Finally, section 7.9 discussed how the realized reference architecture enables *value semantics* through four main properties. First, it features a determinism state machine behavior, where writes generate a new immutable version of a key, identified by a hybrid timestamp, enabling the perception of each key's values in time. Secondly, by versioning logs and using hybrid timestamps, we can observe the value of a key in time and audit the state of a partition at a given timestamp or using physical time. Thirdly, ROTs capture a snapshot of the system, enabling users to perceive the system at a given time. Finally, atomic operations allow a key's state to transition from one value to another.

Chapter 8

Verification and Validation

Contents

The previous chapters described the proposed reference architecture for read-heavy systems and a prototype implementation that realizes this architecture. This chapter verifies and validates our hypothesis, addressing **RQ4** to **RQ6**.

First, section 8.1 describes the methodology used in the verification and validation process. Sections 8.2 and 8.3 present a high-level argument on how the reference architecture achieves CC+ and provides performance-optimal ROTs. Then, section 8.4 describes the empirical validation of the prototype, featuring a comparison of the prototype with its base system, assessing its scalability, and discussing possible threats to the validity of our results. Section 8.5 demonstrates how *value semantics* can be leveraged to audit the system's data at a specific point in time. Section 8.6 summarizes the previous topics.

8.1 Methodology

To verify our hypothesis, in sections 8.2 and 8.3, we discuss how the reference architecture guarantees CC+ and provides performance-optimal ROTs, addressing **RQ4** and **RQ5**.

Then, following an empirical approach, the first phase of the validation (section 8.4.1) features a comparison of the prototype system with its base storage service (Amazon S3). This experiment contributes to **RQ5** by assessing the trade-offs of upgrading the base system with CC. The second phase of the validation (section 8.4.2) focuses on the scalability of the prototype and its underlying

architecture, assessing how it behaves under read-heavy workloads and with an increasing number of read nodes and partitions.

Finally, to address **RQ6**, section 8.5 experimentally demonstrates how enabling *value semantics* facilitates auditing the system's data at a specific point in time.

The scripts used to perform the empirical validation and the experimental verification of the prototype are available in this work's replication package [30], together with the source code and the instructions to replicate each experiment.

8.2 Consistency Guarantees

This section provides a high-level argument on how the selected reference architecture achieves CC+. To that end, it describes how the architecture assures each of the session guarantees expressed in section 2.1.3.1, how it ensures convergence, and discusses how a ROT always returns a causally consistent snapshot of the data.

Read-your-writes. When a client issues a write request, the write node returns the timestamp used to version the client's write, which the client stores in his cache. If the client issues a read request for the same item, and the read node's `stableTime` already reflects the client's write (i.e., is greater or equal to the write's timestamp), then the client observes his write or a more recent write. Otherwise, the client library returns the client's most recent write, which it gets from the client's private cache.

Monotonic-reads. Clients are sticky to a read node (**A7**), and read nodes always return the last value of each item that is already stable (i.e., the version with the greatest timestamp that is lower or equal to the `stableTime`). Therefore, consecutive read requests would only return non-monotonic values if: **i)** the `stableTime` was non-monotonic or **ii)** the read node applied versions with lower timestamps than the `stableTime`. A single write node sequences the writes to one partition in its log in order of arrival using a monotonic clock. Each log is versioned with the clock value of the write node and reflects previous logs. Read nodes only update their `stableTime` after processing the logs of all partitions, and, for each partition, they always fetch a log with a more recent version than the last one they have seen for that partition. The `stableTime` is the minimum of the timestamps of the logs of all partitions, so it always progresses monotonically (a stale log will not be processed after a newer one has been applied), and read nodes will never see a recent log that includes non-applied versions behind the `stableTime`. Thus, neither (i) nor (ii) holds, and the architecture ensures monotonic-reads.

Write-follows-read. By assumption **A8**, clients can only read and update versions from the partitions of their local read node. Thus, if a client observes a version, it must be stable in its local read node. So, for each partition, the read node must already have processed a log version with a timestamp at least as high as the write observed by the client. This implies that the current clock

value of all write nodes is at least as high as that write's timestamp. As the clock advances on each write, regardless of the write node the client writes to, the new write will have a higher timestamp than the write the client initially observed in his read.

Monotonic-writes. If a client performs two writes, the second will always be timestamped before the first. In a first scenario where both writes target items from the same partition, the same write node timestamps them in arrival order and may persist them in the same or consecutive logs. In the first case, read nodes process them in order. In the second case, the log with the lowest timestamp will never be read after the one with the highest because read nodes always fetch monotonically increasing log versions of each partition.

In a second scenario where different write nodes order the writes, the client attaches the timestamp of his last write (`lastWriteTimestamp`) to the second write request, enabling the write node to ensure the second write gets a higher timestamp than the first. In this case, even if the read node gets the log with the second write before the one with the first, it will not make it visible until it has received a log from the first write node because it only advances the `stableTime` when it knows it has seen all versions before that time from all partitions. So, other clients will always observe the writes in their original order.

Convergence. Convergence is guaranteed in all cases because a single write node orders the writes for an item using the last-writer-wins rule, avoiding conflicting writes. Clients may see different versions when reading from their cache but provided that they keep issuing ROTs, their request will eventually retrieve a `stableTime` that already reflects the new version. When that happens, the client library prunes the old version from the cache and returns the most up-to-date version that it received from the read node.

ROTs always return a causally consistent snapshot. ROTs are performed from the snapshot defined by the read nodes' `stableTime`, a timestamp below which it has applied all versions from all partitions. If X is within the snapshot of a ROT, then all its causal dependencies on versions of the same partition were timestamped by the same write node in order of arrival and thus have a lower timestamp than X. Dependencies on versions from other partitions are also guaranteed to be in the snapshot because i) if the same client generated them, the `lastWriteTimestamp` sent on each request guarantees monotonic writes, whether ii) if other clients generated them, then they were already read from a previous `stableTime`, and the `stableTime` increases monotonically.

8.3 Performance-Optimal ROTs

One of the main goals of this dissertation was to produce a reference architecture for read-heavy systems that manifested the *NOC properties* of ROTs: non-blocking, one-round communication, and constant metadata. In this regard, this section discusses how the proposed reference architecture enables each of these properties.

Non-blocking. In each read node, the `stableTime` identifies a stable snapshot across all partitions the read node tracks. A stable snapshot with timestamp ts includes all versions with a timestamp lower or equal to ts, indicating that every write with a timestamp lower or equal to ts has already been applied in that read node. Hence, a ROT can be performed from a stable snapshot without blocking. The client-side cache stores the versions written by the client not yet included in this snapshot, allowing clients to observe monotonically increasing snapshots. This strategy was based on Wren [84] and PaRiS's [85] Client-Assisted Nonblocking Transactional Reads (CANToR) transactional protocol "in which the snapshot of the data store visible to a transaction is defined as the union of two components: i) a fresh causal snapshot that every partition within the DC has installed; and ii) a per-client cache, which stores the updates performed by the client that are not yet reflected in said snapshot"[84, p. 1]. However, given that, in our reference architecture, clients perform all ROTs by contacting the same read node, the snapshot can be computed within each read node. Overall, to ensure non-blocking ROTs, the trade-off is to return a slightly stale but stable snapshot that consists of the union of a stable causal snapshot identified by the read node and a client-side cache.

One-round. According to PORT's definition, "a read-only transaction algorithm has one-round communication if it uses exactly one parallel round of on-path messages and does not have any off-path messages. This matches the messages of simple reads: the client sends a single request to each server holding relevant data, and each server sends a single response back" [58, p. 335].

In the proposed architecture, each read node is responsible for all partitions of its region, and clients are restricted to the items available in their local read node. Thus, a ROT will only require one on-path round of communication to that read node. More precisely, the client sends a single ROT request to its local read node, which computes the transaction's snapshot without requiring any communication with other nodes and returns a single response with the requested versions.

Regarding off-path communication, PORT's [58] definition says that "a message is an off-path message for read-only transactions if its removal affects only the correctness of read-only transactions" [58, p. 335]. However, this definition assumes reliable in-order replication, which is not the case in our reference architecture, where an eventually consistent storage service may be handling write propagation. Thus, no ordering guarantees can be assumed. Consequently, a write node must learn about the minimum `stableTime` across read nodes to perform a checkpoint, which requires additional communication. We would not need the extra checkpointing round for correctness if we assumed reliable in-order delivery. Furthermore, checkpointing communication is not only necessary for ROTs but also for non-transactional reads, which emphasizes the idea that the messages used for checkpointing are a consequence of the delivery mechanism. For the reasons above, we do not consider the checkpointing communication off-path for ROTs.

Constant Metadata. To perform a ROT, a client only needs to specify the keys he wants to read. The response to his request only includes the values of the requested keys and the `stableTime` at the read node. Thus it does not increase with the size of the system.

8.4 Empirical Validation

In this section, we empirically evaluate the prototype system against a set of metrics identified in section 3.4, such as read and write latency and throughput, data staleness, and goodput.

Firstly, we compare the prototype system against its base system, Amazon S3, identifying the trade-offs that arise from upgrading its consistency guarantees. Then, to validate the suitability of the prototype for read-heavy systems, we evaluate the latency and throughput of the system under read-heavy workloads and assess how it scales with the number of nodes and clients.

The workloads used for this validation were generated from scratch, enabling us to control the validation environment, avoid randomness, and easily detect flaws. When aiming to validate read latency, throughput, and visibility, we used a constant write generator, where each client thread uses a client process to generate write requests at a fixed rate, and a busy write generator, where each client issues read requests in a closed loop, as fast as possible. When validating goodput and write throughput, we used a busy write generator, where each client thread generates write requests in a closed loop, and a constant read generator, where clients issue read requests at a fixed rate. When using multiple clients to generate requests, we deployed the load generator in an Amazon EC2 instance, enabling each client thread to execute in a separate core. Given the prevalence of small items in several production workloads [6], the payload of each write corresponds to an 8-byte long (12 bytes when encoded for transfer). Overall, this strategy aimed to ensure that when measuring read performance, the write load was as constant as possible and vice-versa.

In order to avoid bursts of checkpointing and garbage collection activity, especially given the short time frame of each experiment, we deactivated these for all experiments.

The scripts used to perform this empirical validation are available in this work's replication package [30], together with the source code and the instructions to replicate each experiment.

8.4.1 Comparison with Amazon S3

To show the trade-offs of providing CC atop Amazon S3, we modified our codebase and architecture, removing the compute layer and the client-side logic used to guarantee causality. In the baseline system, clients issue requests directly to Amazon S3, providing it with the necessary conditions to deliver the best possible performance.

Given the pay-per-request policy of Amazon S3 and that clients of the baseline system issue their requests directly to Amazon S3 buckets, these first experiments were conducted on a simple setting: two AWS regions (Ireland and North Virginia), a single data partition replicated in both regions and two clients per region, one issuing read requests and the other issuing write requests.

For each system component, we provisioned a t4g.small Amazon EC2 instance (2vCPU and 2GiB of RAM), keeping the costs low for anyone who wants to replicate these experiments. For the prototype system, we set up a read node in each region and a write node in Ireland. In both systems, one client issues write requests in Ireland, and two clients issue read requests, one in each region. The diagram of fig. 8.1 illustrates the deployment of the baseline system, and fig. 8.2 shows the deployment of the prototype system.

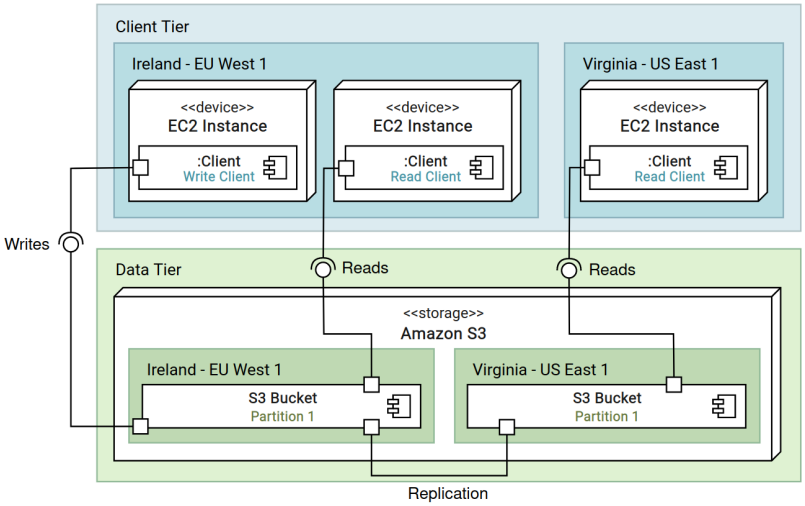


Figure 8.1: Baseline system validation infrastructure UML deployment/component diagram. One Amazon EC2 instance was deployed for each client. Clients issue requests to the bucket of their region. A client process per region issues read requests, and one in Ireland issues write requests. Amazon S3 handles replication.

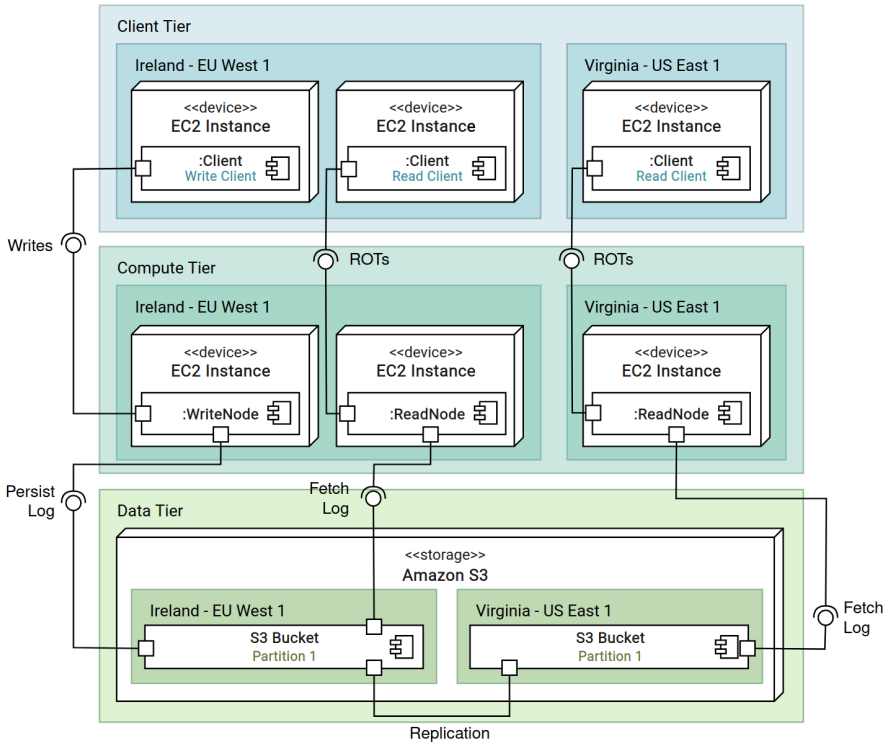


Figure 8.2: Prototype system validation infrastructure UML deployment/component diagram. One Amazon EC2 instance was deployed for each client and server. Clients issue requests to the servers of their region. One client process per region issues read requests, and one in Ireland issues write requests. Write and read nodes persist and fetch the log from Amazon S3. Amazon S3 handles replication.

Clients issue read and write requests for a single key. In the prototype system, the log is persisted and fetched every five seconds. After each experiment, we emptied all Amazon S3 buckets, extracted the generated logs, and destroyed the docker containers.

In the first round of the experiment, the reading clients issue requests in a closed loop, whereas writing clients issue requests at a fixed rate. We only accounted for new reads (i.e., reads that retrieved a new version of the item), and only the last 100 observations of each test were considered. We measure read and write latency in the local region of the write client and staleness in both regions. The experiments were repeated for different inter-write delays (50, 100, and 200ms).

In the second round of the experiment, read clients read at a fixed rate for 20 seconds and write clients perform requests in a closed loop. We measure the system's goodput (i.e., the number of versions that transverse the system from a writing client to a reading client in a second). This experiment was repeated thrice for different inter-read delays (50, 100, and 200ms).

8.4.1.1 Results

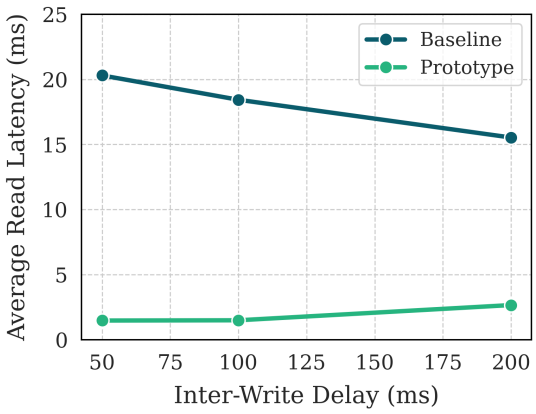
Considering the relevancy of read performance in read-heavy systems, these experiments aim to measure the impact of upgrading Amazon S3 's consistency guarantees on the latency of read requests. Moreover, taking into account the trade-off between performance and visibility observed in the literature [85, 8], especially in more recent stabilization-based protocols, they aim to understand how this decision impacts visibility.

The following results derive from the two experiment rounds outlined earlier: latency and visibility were measured in the first round with a constant inter-write delay, whereas goodput was measured in the second round with a constant inter-read delay.

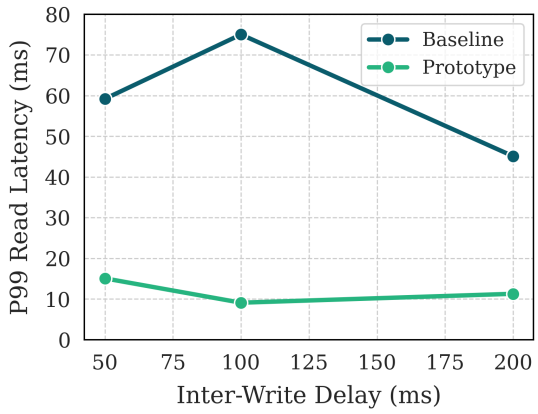
Table 8.1 shows the latency distribution for both systems using a 50ms inter-write delay. As we can observe, the read latency of our prototype implementation consistently outperforms Amazon S3, even though Amazon S3 is destined for frequently accessed data. It is noteworthy to mention that this result was not an isolated occurrence. One factor that may have impacted the result is client-server communication. In the prototype, we use gRPC, which relies on HTTP/2, providing reduced latency over HTTP/1.1, and leverages Protocol Buffers for efficient data serialization. In the baseline, the communication is done directly with Amazon S3 through AWS 's SDK for Java, which uses HTTP/1.1. The separation of read and write load in the prototype system may be another factor influencing these results because, even though the clients target the same key, in

System	maximum	99%	95%	70%	50%	average	minimum
Baseline	79.00	59.20	51.05	19.00	16.00	20.30	10.00
Prototype	22.00	15.07	6.10	1.00	1.00	1.47	< 0.01

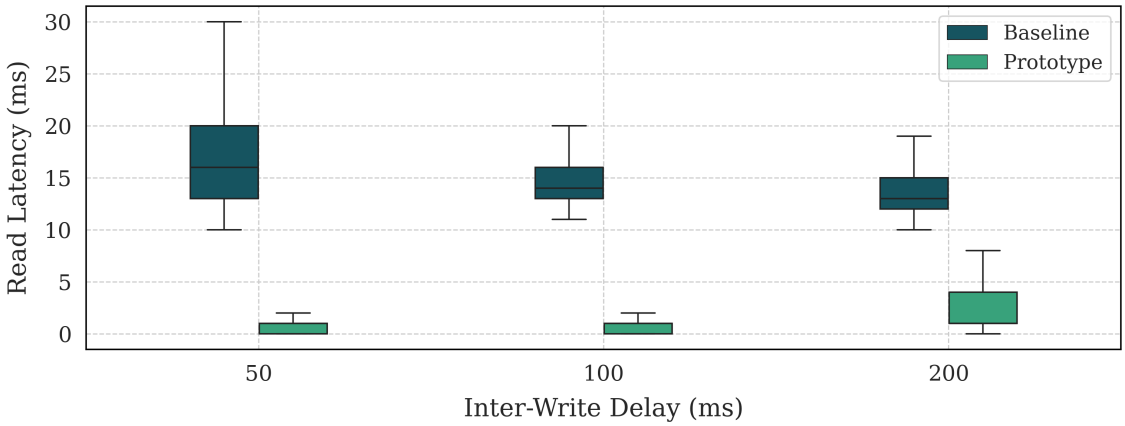
Table 8.1: Read Latency (in ms) in the baseline and prototype systems for an inter-write delay of 50ms. The results suggest lower read latency in the prototype, which may result from the communication protocol, the separation of write and read load, or other implementation factors.



(a) Average Read Latency (in ms)



(b) 99th Percentile of Read Latency (in ms).



(c) Read Latency (in ms) boxplot (without outliers). Each box represents, from bottom to top, the minimum, first, second and third quartiles and the maximum latency.

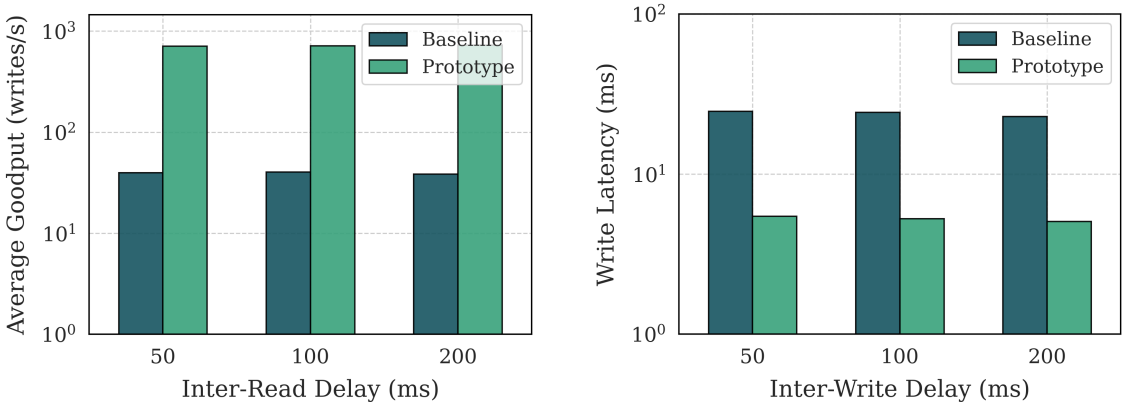
Figure 8.3: Read Latency (in ms) for the baseline and prototype systems using different inter-write delays (50, 100, and 200ms). The prototype's latency consistently outperforms the baseline's, possibly due to the communication protocol, the separation of write and read load in the prototype or other implementation factors. Figures 8.3a and 8.3b suggest that the average and 99th percentile latency improves with higher delays in the baseline system. The prototype's average latency does not exhibit a significant variation, but the 99th percentile slightly improves. Without outliers (fig. 8.3c), the prototype's latency increases for a 200ms delay.

the prototype, the write load is handled by the write node and the read load by the read node. In contrast, in the baseline system, all requests target the same Amazon S3 key.

With CC, 70% of the read requests present a latency of less than 1ms, and 99% take less than 15.07ms, while, in the same setting, the baseline presents a 70th percentile read latency of 19ms and a 99th percentile of 59.20ms.

Figures 8.3a and 8.3b show the average and 99th read latency for different inter-write delays. As expected, with higher inter-write delay, the baseline shows an improvement in average and 99th percentile latency, which probably results from lower contention on the key that is being accessed. The prototype's results for the average latency do not exhibit a significant variation with the inter-write delay, whereas the 99th percentile exhibits a slight improvement. Figure 8.3c summarizes the

latency results in a boxplot, excluding the outliers. Even though the results for a 200ms inter-write delay suggest a slight latency degradation for the prototype, the 99th and average latency do not exhibit a relevant difference, which leads us to dismiss this result.



(a) Average System Goodput (in writes/s).

(b) Average Write Latency (in ms).

Figure 8.4: Average System Goodput (in writes/s) and Write Latency (in ms) of the baseline and prototype systems using different inter-read delays (50, 100, and 200ms). The prototype exhibits higher goodput and lower write latency because it only persists writes periodically, while in the baseline, write requests are synchronous and only return when the write is durable.

Figure 8.4a shows the systems' goodput in writes per second for different inter-read delays in logarithmic scale. Both systems display a significant discrepancy because, in the baseline, the writes are forwarded directly to Amazon S3, which, to guarantee durability, will only return a response when the new item has been replicated across multiple availability zones. In contrast, in the prototype, writes are persisted periodically and asynchronously, avoiding synchronous calls on each request. Moreover, as we observed for read latency, we reduce the contention resulting from simultaneous writes and reads performed for the same key by separating read and write load. This claim is further supported by the results presented in fig. 8.4b, which exhibit lower write latency for the prototype.

As expected, the read performance, goodput improvements, and additional consistency guarantees come at the cost of increased data staleness, as observed in fig. 8.5. As write requests are batched in the operation log and only persisted after Δs (5s in this case), the time it takes for a new version to be persisted in Amazon S3 is higher in comparison with the baseline, where writes are immediately persisted to the respective Amazon S3 bucket. Moreover, even though this experiment uses a single partition, in case of multiple partitions, a version will only be made visible when it is stable. However, due to clock synchronization, this time is bounded. Therefore, the determinant factor in the increased staleness is the log propagation strategy. On the other hand, this strategy offers the possibility to balance this trade-off by managing the frequency with which logs are persisted at write nodes and with which read nodes update their cache. Regardless, in remote regions, the impact of providing CC on visibility is less notable.

In light of these results, we highlight that the cost of upgrading existing cloud storage services

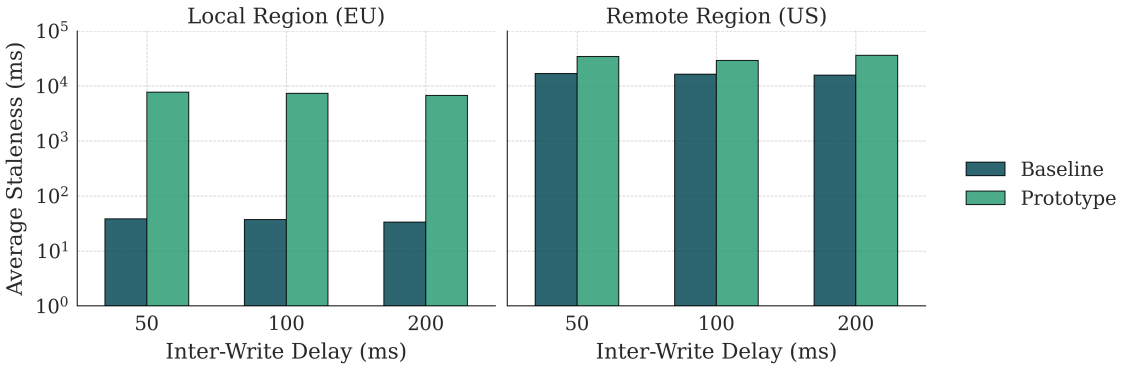


Figure 8.5: Average Staleness (in ms) of the baseline and prototype systems using different inter-write delays (50, 100, and 200ms). On the left, the figure shows the average staleness in the write's origin region. On the right, it shows the staleness in a remote region. Staleness is higher in the prototype because writes are batched and persisted periodically. The discrepancy between the baseline and the prototype's staleness is lower in remote regions.

with CC is the increased data staleness, especially within the region where the write was initially generated. However, changing the frequency of log pushes and pulls can control this trade-off. The results suggest that the client and server-side logic for guaranteeing CC incur negligible performance overhead. Moreover, they indicate that the proposed architecture, particularly its log propagation strategy, can result in lower read latency and higher goodput.

8.4.2 Scalability

This set of experiments aims to study the system's applicability, and more importantly, of its underlying architecture, to read-heavy systems and evaluate its scalability.

We used the same replication sites (Ireland and North Virginia) for all experiments and co-located clients with their local read nodes. We spawn two client processes in Ireland, one issuing read requests and another issuing write requests. For the visibility tests, we provisioned another reading client process in North Virginia. To generate different load conditions, we spawn a different number of threads per client process.

We provisioned a t4g.small Amazon EC2 instance (2vCPU and 2GiB of RAM) for each server (read and write nodes) and used a c6g.8xlarge Amazon EC2 instance (32 vCPU and 64 GiB of RAM) for each client process. The deployment of this experiment is similar to the one described in the previous section and is illustrated in fig. 8.2. However, we used different Amazon EC2 instances for client processes to enable more client threads.

The log persistence and fetch rates were set to five seconds in all the tests.

The first test assesses read latency, throughput, and visibility in a read-heavy workload. Read clients issue ROTs in a closed loop for 60 seconds in the write's origin region and 90 seconds in the remote region. Each client performs two reads per ROT from a set of 8 keys (i.e., looping through the list of keys two by two). Write clients issue 1000 write requests at a fixed rate (every 50ms) using an 8-byte payload. When measuring latency, we only account for new reads (i.e.,

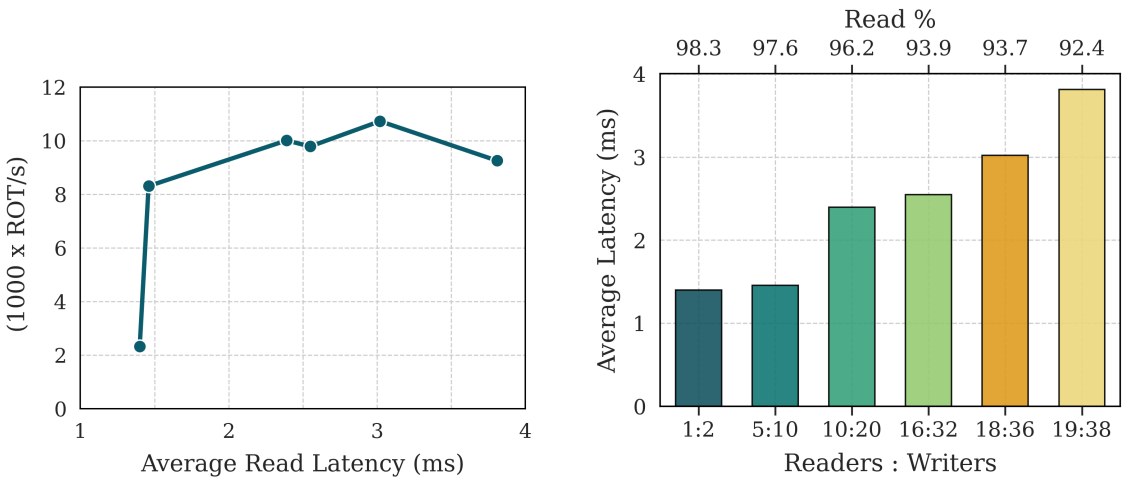
reads that retrieved a new item version). To simulate a read-heavy workload in this setting, we required more writing than reading clients because of the different request rates. In particular, we used double writers than readers, resulting in different read:write ratios with read prevalence ranging from 92% to 98%. The measurements were repeated with increasing client threads until the system was saturated (i.e., adding more client threads resulted in lower read throughput).

Then, using a fixed number of read and write clients (five readers and ten writers), we repeated the measurements with varying partitions (1, 2, and 4).

We used a similar setting in a third experiment but evaluated how the system scales with the number of read nodes, clients, and partitions. First, for an increasing number of read nodes (1, 2, 3, and 4), we used a fixed number of writing clients, each issuing 500 requests at a fixed rate, and measured read throughput for an increasing number of reading clients (1, 5, 10, 15, 20, and 25), until the system got saturated. Read clients issue read requests for 30 seconds for the same set of 8 keys, performing two reads per ROT. Then, to test how the system scales with the number of partitions, we used a write-only workload with varying clients issuing write requests in a closed loop for 30 seconds and measured write throughput for 1, 2, and 4 partitions.

8.4.2.1 Results

Figure 8.6a shows the read performance results for different numbers of client threads, each represented by a "dot" in the plot. We can observe the nearly ideal improvement in read throughput with almost no penalty in read latency when moving from the minimum number of client threads



(a) Read Performance with different numbers of client threads.

(b) Average Read Latency (in ms) with different numbers of client threads.

Figure 8.6: Read Performance with different numbers of client threads. On the left, we can observe the machine's saturation point around 10K ROTs. Each "dot" corresponds to a different number of client threads. On the right, we can see the average latency for the same number of client threads and the corresponding read percentage. The average latency keeps below 4ms even in overload conditions.

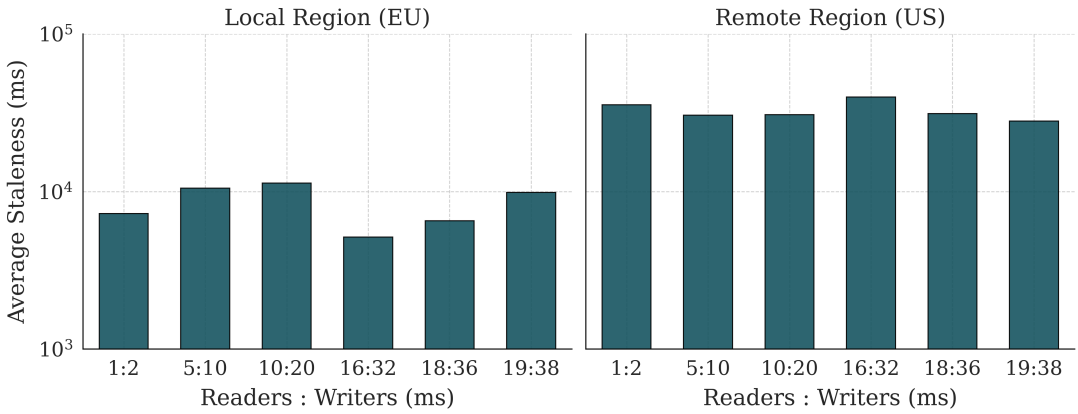
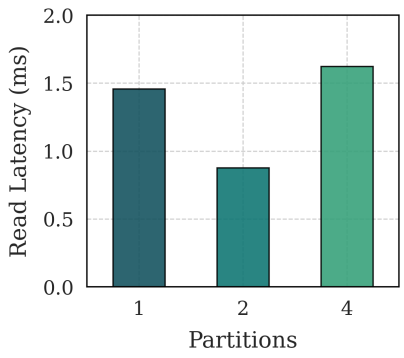


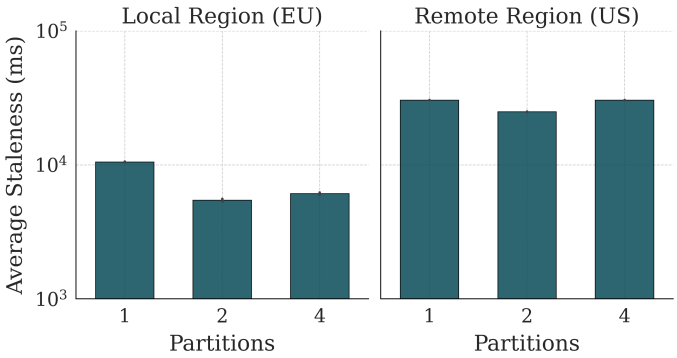
Figure 8.7: Average Staleness (in ms) for different load conditions in logarithmic scale. The load increase does not affect data visibility. On the write's origin region, the average staleness is around twice the rate at which the log is pushed and pulled to Amazon S3. In the remote region, it is slightly higher due to the replication delay.

(one reader and two writers) to a setting with five times more read and write load. As more load is added, the latency increases more rapidly, and read throughput decreases around 10K ROTs per second, representing the machine's limit for this particular setting. Figure 8.6b shows the average latency for this experiment and the percentage of read requests for each number of client threads. From the read-and-write ratio, we can observe a higher drop in the rate of reads when adding new threads after the system is saturated. The average latency keeps below 4ms even in overload conditions and when using a small-sized instance with limited capacity.

For the same setting, fig. 8.7 shows that the load increase does not affect data visibility. In the local region, the average staleness is around 10 seconds for all workloads, twice the rate at which the log is pushed and pulled to Amazon S3. In the remote region, it is slightly higher due to the replication delay. These results suggest that, even though staleness is the price to pay for increased



(a) Average Read Latency (in ms) for a different number of partitions.



(b) Average Staleness (in ms) in logarithmic scale for a different number of partitions.

Figure 8.8: Average Read Latency and Staleness for a different number of partitions. Both results show that the number of partitions does not affect read latency or visibility.

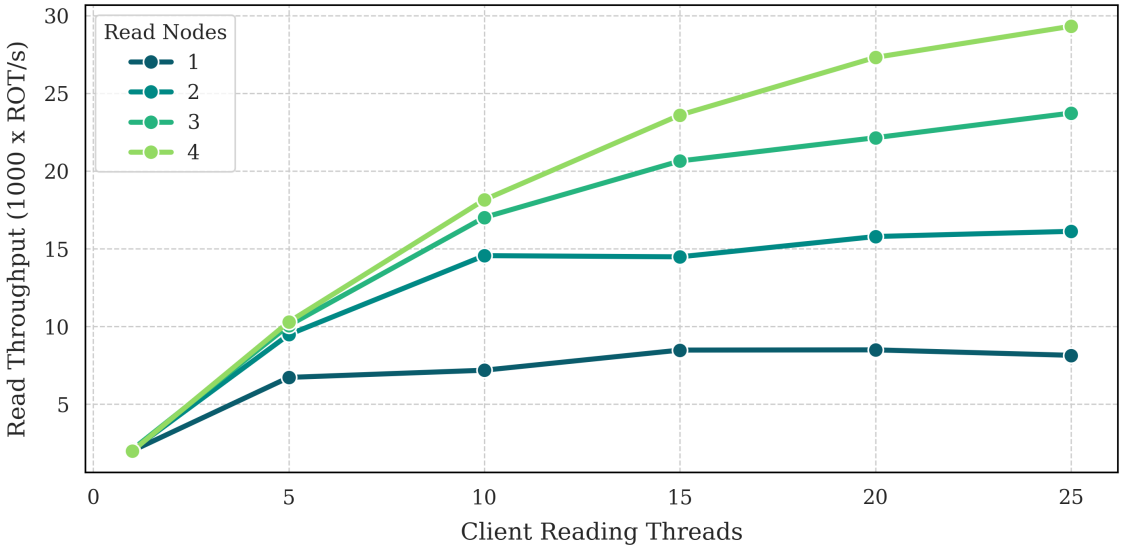


Figure 8.9: Read Throughput (in ROT/s) for a different number of read nodes. The system's read capacity increases almost linearly with the number of available read nodes.

consistency, it is bounded by log upload and retrieval rates.

Figures 8.8a and 8.8b show that the average latency and staleness keep approximately stable when varying the number of partitions. This result is expected because all write nodes periodically persist the log at the same rate, the only difference being that with more partitions, read nodes must fetch more logs. In terms of latency, it shows how separating writes and reads allows read performance to be independent of how the write load is distributed.

Figure 8.9 reports the read throughput achieved with varying read nodes. As expected, the system's read capacity increases almost linearly with the number of available read nodes. These results confirm the system's scalability upon increased read load, affirming the suitability of its architecture for read-heavy systems.

Finally, fig. 8.10 shows the write throughput achieved with 1, 2, and 4 partitions. The write capacity increases almost linearly with the number of partitions, which is expected given that, as the number of partitions increases, the items targeted by the writes nodes will become more and more spread across partitions, thus decreasing the load in each write node. Even though the reference architecture targets read-heavy systems, these results demonstrate that having a write node sequencing the writes per partition also enables the write load to be balanced according to the partitioning strategy.

8.4.3 Threats to Validity

Considering the empirical nature of the validation process, it is important to consider the threats to the validity of its results [27]. In this regard, this section analyzes the internal (i.e., the possibility that the conclusions observed were affected by other factors) and external validity (i.e., the extent to which the results can be generalized to other contexts) of the present research.

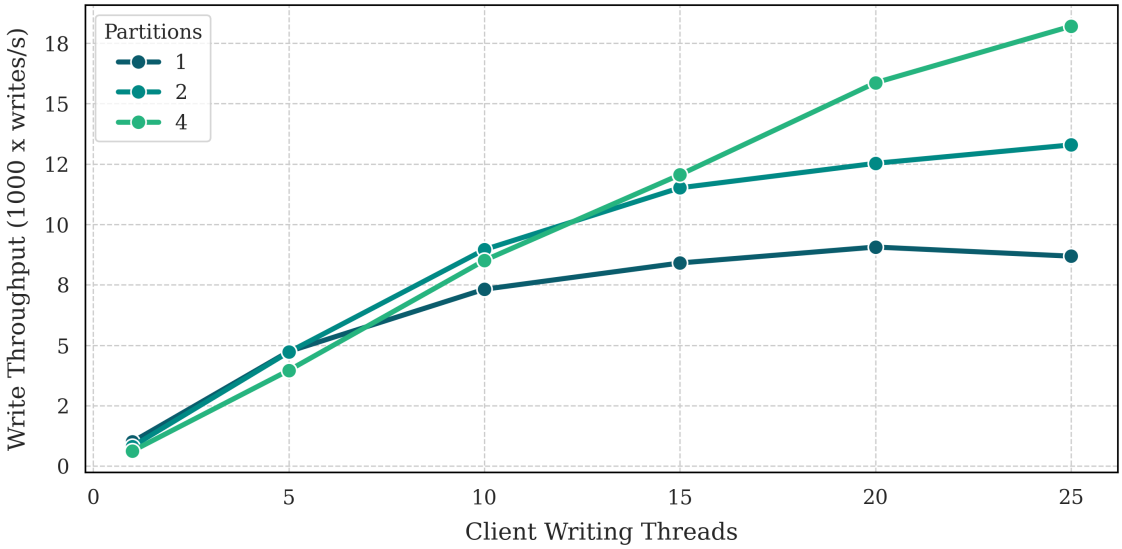


Figure 8.10: Write Throughput (in writes/s) for different number of partitions. The write capacity increases almost linearly with the number of partitions.

8.4.3.1 Internal Threats

The experiments were performed on different Amazon EC2 instances.

During the validation process, experiments were conducted iteratively and occasionally repeated when bugs were detected. Moreover, instances were terminated upon each experiment to minimize resource usage and costs. Therefore, there may be slight performance discrepancies between experiments.

The start-up time of the client's containers is beyond our control.

The start-up time of the client's containers is beyond our control and may introduce discrepancies between test executions. In particular, when evaluating goodput within short experiments, even a minor disparity in the start-up time of the client's container may result in a significantly different result because the log gets pushed periodically. The disparity between executions may be enough for a set of versions only to be persisted in the subsequent execution of the thread that persists new versions to Amazon S3. In order to mitigate this limitation to the best extent possible, goodput experiments were repeated three times.

8.4.3.2 External Threats

The reference architecture was realized in a single storage service.

Due to time constraints, the reference architecture was only realized within the scope of Amazon S3, a NoSQL key-value cloud storage service that provides read-after-write consistency. Therefore, its validity when implemented on top of other cloud storage services that offer pure EC or stronger semantics is yet to be tested in future research. If realized in a purely eventually consistent storage service, we expect the baseline system to deliver better

write latency, system goodput, and visibility, and thus the impact of providing CC may be more significant concerning these metrics. If applied to other storage services, it may be necessary to rethink the strategy used to index the logs by timestamp, which will depend on the data model and provided interface.

The prototype's servers were deployed in low-cost Amazon EC2 instances.

In order to reduce the costs of reproducing our findings and make it feasible to observe the prototype's behavior with small configuration and workload modifications, read and write nodes were deployed using free-tier t4g.small instances. With fewer vCPUs, the system's capacity for concurrently processing ROTs and its ability to persist and retrieve the log in parallel is constrained, potentially resulting in higher read latency and lower throughput. If the experiments described in sections 8.4.1 and 8.4.2 are repeated in more powerful instances, an overall improvement in the prototype's performance is to be expected.

The prototype's components were launched on a logically isolated virtual network.

The prototype was deployed and validated using AWS cloud infrastructure, where resources run on an isolated virtual network. Therefore, the validation results (e.g., latency, visibility, goodput, and throughput) may differ if the reference architecture is realized and assessed using other cloud providers or in a production environment.

Amazon EC2 instances virtualize the hardware.

Even though AWS strives to minimize volatility, hardware virtualization in Amazon EC2 instances may introduce slight discrepancies in the performance between experiments, potentially impacting the generalizability of the results to other execution environments.

The empirical comparison against the baseline system was limited to a two-client setting.

In order to assess the overhead of upgrading Amazon S3 's consistency guarantees, the prototype implementation was directly compared with the base system. Despite enabling the base system to deliver the best possible performance, this approach constrained the comparison under heavier workloads because clients issue the requests directly to Amazon S3 buckets. Due to Amazon S3 's pay-per-use service model, repeating this experience with an increasing number of clients became impracticable, particularly since a subset of clients issue requests at the maximum possible rate. However, repeating this comparison at scale would complement our analysis, providing valuable insight into the impact of upgrading Amazon S3 with CC under heavier load conditions.

The selected testing scenarios and workload configurations do not cover all possible load conditions.

Even though we validated the prototype for various load conditions (e.g., varying inter-operation delays, number of clients, read and write ratios, and number of system nodes), testing all relevant load conditions and system configurations was unfeasible due to cost and time constraints. Thus, different results may be observed in untested scenarios or load conditions. In particular, we did not assess the prototype in skewed workloads, with more than

two keys per ROT, or with different payload sizes. Furthermore, the prototype was tested for only two regions because the clients always contact their local read node, and thus adding new regions does not influence the system in terms of read performance. Regarding write latency and staleness, the differences would only result from Amazon S3 's propagation delay between regions and the distance of the clients to write nodes.

8.5 Value Semantics

This section addresses **RQ6** by demonstrating how *value semantics* can be leveraged to audit the system's data at a specific point in time.

```
$ python3 . -k a,b -t 0000000168709110000-0000000000000000000

Key = a
 > value = 274877906968
 > timestamp = 0000000168709108-0000000000000000006
 > log version = 0000000168709110-0000000000000000001
 > partition = 2

Key = b
 > value = 274877906969
 > timestamp = 0000000168709108-0000000000000000007
 > log version = 0000000168709110-0000000000000000002
 > partition = 1
```

Listing 8.1: Value semantics experimental validation — Observing the state of a set of keys at a given timestamp. The command returns the value and timestamp of keys *a* and *b*, each belonging to a different partition, at the specified timestamp and the log version where those versions were first observed.

To begin, we deployed the prototype system on Amazon EC2 using a setting with a single region, two partitions, and two clients alternating write requests between partitions. We set up two write nodes in Ireland and two clients in the same region. Each client issues write requests for approximately 5 minutes at a constant rate, with a 3-second delay between each write. Clients alternate between a set of four keys, two from each partition. To make it easier to observe the state changes of each key, each client issues the first request with the payload "274877906944" and increments this value for each write. As in our empirical validation, checkpointing and garbage collection were deactivated for this experiment.

After running the experiment, we extracted the operation logs from the respective Amazon S3 buckets. Then, we developed a Python script that, given a set of keys and a date or timestamp, retrieves the versions of the specified keys at the given time or shows the progression of values of a given key in time. Listings 8.1 to 8.4 illustrate the four usage scenarios of this script.

The first scenario, depicted in listing 8.1, demonstrates the ability to observe the version of a set of keys, possibly distributed across partitions, at a specific timestamp. This can be done almost linearly because logs are versioned with hybrid timestamps, making it possible to determine the lowest log version that must be parsed (i.e., the lowest log version whose timestamp is at least as high as the provided timestamp). If checkpointing and garbage collection of Amazon S3 logs were enabled, there might not be a version available to display because it might already have been removed. This highlights the need to balance the trade-off between storage space and auditability by managing the periodicity with which a checkpoint is performed and the number of stale versions kept in Amazon S3 buckets.

```
$ python3 . -v -k a,b -d "2023-06-18 13:25:00.0"

Key = a
 > value = 274877906968
 > timestamp = 0000000168709108 9-0000000000000000006
 > log version = 0000000168709110 9-0000000000000000001
 > partition = 2

Key = b
 > value = 274877906969
 > timestamp = 0000000168709108 9-0000000000000000007
 > log version = 0000000168709110 9-0000000000000000002
 > partition = 1
```

Listing 8.2: The command returns the value and timestamp of keys *a* and *b*, each belonging to a different partition, at the specified date-time and the log version where those versions were first observed.

In the second scenario, illustrated in listing 8.2, a human-readable date-time format is used instead of a timestamp, confirming the benefit of using hybrid timestamps to sequence the write operations, especially in terms of intuitiveness and ease of auditing.

```
$ python3 . -hist -k a -t 0000000168709102 9-0000000000000000005

History of key a:
  timestamp value
0 0000000168709102 9-0000000000000000000 274877906944
1 0000000168709102 9-0000000000000000001 274877906944
2 0000000168709102 9-0000000000000000004 274877906948
3 0000000168709102 9-0000000000000000005 274877906948
```

Listing 8.3: The command returns the ordered set of versions (timestamp and value) of key *a* up to and including the provided timestamp.

```
$ python3 . -hist -k a -d "2023-06-18 13:24:00.0"

History of key a:
  timestamp value
0 00000001687091029107-00000000000000000000 274877906944
1 00000001687091029107-00000000000000000001 274877906944
2 00000001687091029107-00000000000000000004 274877906948
3 00000001687091029107-00000000000000000005 274877906948
```

Listing 8.4: The command returns the ordered set of versions (timestamp and value) of key *a* up to and including the provided date-time.

In the remaining scenarios, illustrated in listings 8.3 and 8.4, given a key and a timestamp (or a human-readable date-time format), it is possible to observe the succession of values of a given key, enabling developers to analyze and verify the evolution of the key's values throughout the system's operation. This operation can be performed by only fetching a single log, which contains all the history until that point. If, however, checkpointing and garbage collection were enabled, it might be necessary to fetch all the logs before the provided timestamp.

Overall, this experiment demonstrates how enabling value semantics, mainly by having immutable versions of each key versioned with a hybrid timestamp and sequenced in logs, significantly improves the auditability of the system by making it possible (1) to reason about the state of the system at a given point in time, (2) to observe the progression of values for a given key, and (3) to use physical time to audit the system intuitively. However, it also highlights the need to balance the trade-off between storage space and auditability, specifically by controlling the periodicity of checkpoints and by preserving a large enough number of log versions.

8.6 Summary

In this chapter, we have covered this work's validation and verification process.

Section 8.1 described the methodology adopted. It combined a high-level argument on how the reference architecture provides the desired properties, an empirical validation of the prototype concerning a set of metrics, and an experimental verification to demonstrate how enabling *value semantics* facilitates auditing.

Section 8.2 discussed how the reference architecture ensures each of the session guarantees that characterize CC, convergence, and ROTs by combining a client-side cache, with log versioning and client and server-side logic.

Then, section 8.3 outlined how the proposed architecture enables the *NOC properties* of ROTs. The CANToR transactional protocol [84] was leveraged to ensure non-blocking ROTs. We avoided a coordinator-based approach to enable one-round communication between client and server by restricting clients to their local read node, which tracks all partitions of its region. This strategy and the usage of hybrid timestamps also enabled constant metadata.

Section 8.4 described this work's validation process.

First, section 8.4.1 featured the comparison of the prototype system with Amazon S3, the storage service on top of which it is built. The results of this comparison show that the read latency of our prototype implementation consistently outperforms Amazon S3 and highlight that the cost to pay for upgrading existing cloud storage services with CC is the increased data staleness. This trade-off can be managed by adjusting the frequency with which the log is persisted and fetched from the storage service.

Section 8.4.2 assessed how the realized architecture performs under read-heavy workloads and scales with the number of clients, system nodes, and partitions. The results demonstrated that the average latency and staleness keep approximately stable when varying the number of partitions. Read and write throughput increases almost linearly with the number of available read and write nodes.

Section 8.4.3 pointed out the external and internal threads that may have impacted the validity of the results. External threats primarily relate to constraints introduced by the AWS cloud infrastructure used in the validation process and with possible untested load conditions and configurations. The internal threads identified relate to potential inconsistencies in the environment of each experiment, which may have resulted from using different instances per experiment or from slight delays when starting up each system component.

Finally, section 8.5 demonstrated how enabling *value semantics* in our prototype implementation makes it possible to reconstruct the system state, allowing the developer to reason about its behavior at a point in time or to observe the progression of values of a particular key.

Chapter 9

Conclusions

The previous chapter discussed the verification and validation of our hypothesis. This chapter presents the final remarks of this dissertation (*cf.* section 9.1). Section 9.2 summarizes the main findings regarding each research question. Section 9.3 revisits the hypothesis, describing how our work supported each premise. Section 9.4 outlines the main contributions of this work. Then, section 9.5 presents the main challenges that arose throughout the development of this dissertation. Finally, section 9.6 identifies possible directions for future work.

9.1 Summary

Academic literature has positioned CC as an attractive choice for achieving high availability, performance, and intuitive behavior in geo-replicated systems without the overhead of stronger consistency models or the ordering anomalies of EC. Additionally, it has proposed several causally consistent system implementations, such as the ones surveyed in this work's systematic literature review (*cf.* chapter 3), each featuring its benefits and trade-offs. Most importantly, recent work [58, 85], motivated by the prevalence of reads in real-world applications [15, 18, 69, 92, 100], has focused on improving the performance of ROTs within causally consistent systems.

Despite the relevancy of these novel results in the scope of read-heavy systems, our literature review highlighted the lack of a standard architecture for applying CC to read-heavy systems and the need to integrate existing cloud storage solutions, which offer added benefits in terms of availability, reliability and data accessibility. Additionally, it suggested the need to leverage the auditing properties of existing implementations further.

In light of these shortcomings, in this dissertation, we designed a reference architecture for read-heavy systems (*cf.* chapter 6) that provides CC+ atop existing cloud storage services, manifests performance-optimal ROTs and enables auditing through *value semantics*.

To that end, we first translated the knowledge acquired in our systematic literature review into four system architectures, which served as the starting point for developing the reference architecture. Following an engineering approach, we iteratively built and refined the reference

architecture through the ATAM method. This process culminated in our final reference architecture (*cf.* chapter 5).

The reference architecture was then realized in a prototype system, using Amazon S3 as the underlying storage service (*cf.* chapter 7). The prototype was deployed using AWS's cloud infrastructure and evaluated against a set of metrics: read latency and throughput, data staleness, and goodput. We compared the prototype against its base system, assessing the trade-offs that arise from upgrading its consistency guarantees (*cf.* section 8.4.1). Additionally, we evaluated the latency and throughput of the system under read-heavy workloads and assessed how it scales with the number of nodes and clients (*cf.* section 8.4.2).

The results from our validation showed that the read latency of our prototype implementation consistently outperformed its underlying storage service and highlighted that the cost to pay for upgrading existing cloud storage services with CC is the increased data staleness. This trade-off can be managed by adjusting the frequency with which the log is persisted and fetched from the storage service. They also demonstrated the ability of the system to scale with the number of read nodes, partitions, and clients.

Finally, we experimentally demonstrated how our prototype implementation provides *value semantics*, enabling the developer to reason about the system's behavior at a point in time (*cf.* section 8.5).

In the next sections, we revisit our research questions and hypothesis, explaining how they were addressed in this dissertation.

9.2 Research Questions

In this section, we revisit the research questions that guided this dissertation, showcasing how each question was addressed and summarizing the key findings obtained.

RQ1. What are the ideal properties of a geo-replicated causally consistent read-heavy system?

As described in section 2.2, in geo-replicated causally consistent read-heavy systems, read operations comprise most of the request load, making it relevant to optimize the performance of ROTs, which incur additional coordination overhead than non-transactional reads but are necessary for retrieving a consistent view across shards. In this regard, the literature defines performance-optimal ROTs based on the properties of non-transactional reads (see section 2.2.2), which are non-blocking, complete in a single round of on-path communication, and use constant metadata. Ideally, a causally consistent read-heavy system should support ROTs that exhibit these properties because they capture the minor coordination overhead and the best performance that can be achieved with ROTs.

RQ2. What properties and strategies do existing causally consistent systems employ?

Considering the fourteen causally consistent systems and two architectural approaches to CC surveyed in our literature review (sections 3.2 and 3.3), existing causally consistent systems can be classified according to several properties, some of which being (1) the operations they support, (2) their clock type, (3) the replication model, and (4) the *NOC properties* of their ROT algorithm. Concerning the taxonomy, some systems support generic transactions [98, 4, 65, 85] or WOTs [54, 58], but the majority only supports ROTs. The prevalent mechanism to enforce causality is logical time (e.g., Lamport timestamps or vector clocks), but some protocols rely on HLCs [85, 84, 23, 91] or use physical clocks [5, 24, 25, 4, 86]. The majority of the systems do not support partial replication. Regarding the *NOC properties*, the systems that use physical clocks are blocking. Most algorithms require multiple rounds to complete a ROT, except SwiftCloud [98], which reads from the cache, and Eiger-PORT [58], which reads from a client-defined timestamp. Several systems use one scalar timestamp to track causality; others use vector clocks or exchange dependencies. The analyzed architectures also feature relevant properties: Bolt-on [8] can upgrade an ECDS with CC and is storage-agnostic, and MongoDB [91] integrates a primary-replica strategy with a modified implementation of HLCs that considers an additional requirement, security.

To ensure CC, the systems typically use a strategy that falls within one of these categories: dependency checking, sequencer-based, stabilization, or optimistic. Systems that use dependency checking [53, 54, 24, 56] encode causal dependencies in metadata and verify them before applying an operation. Sequencer-based techniques [5, 98] impose a total order of operations on each replica. Stabilization protocols [25, 4, 23, 84, 85] establish a cutoff timestamp (or set of timestamps) below which all updates from remote DCs have been applied and only make versions comprised in the snapshot defined by this timestamp visible to clients. Finally, in systems that adopt an optimistic approach [86, 56], clients are the ones to enforce causality, and updates are immediately applied, even if their dependencies are not yet available.

RQ3. What metrics have been used to evaluate these systems?

Based on our review (section 3.4), metrics commonly used to evaluate causally consistent distributed systems include latency, throughput, tail latency, scalability, resource overhead, and data staleness. To assess latency and throughput for each operation, the authors compare the results against ping requests, which establish hardware-imposed limits, or with other systems with weaker consistency guarantees. Tail latency, measured using high percentiles like 95^{th}, 99^{th}, and 99.9^{th}, helps identify requests that take longer than the average. Throughput is commonly measured for varying read-to-write ratios, number of keys per operation, and inter-operation delays. To assess how the system scales, authors evaluate the system's throughput for different numbers of DCs, partitions per DC, and client threads. Resource overhead is evaluated through metrics such as metadata exchanged, CPU usage,

and memory usage. Data staleness is measured by calculating the visibility latency of each update.

RQ4. Can we produce a cloud-native reference architecture for read-heavy systems that manifests the performance-optimal properties of read-only transactions? How does it compare with state-of-the-art causally consistent systems?

Drawing upon the review undertaken in **RQ1** and **RQ2** and on our preliminary studies (chapter 5), this work proposes a cloud-native reference architecture for read-heavy systems (chapter 6), compares it with state-of-the-art systems (section 6.2) and confirms its ability to provide performance-optimal ROTs (section 8.3) by incorporating the literature's findings. First, the architecture adapts the CANToR transactional protocol used in Wren [84] and PaRiS [85] to ensure non-blocking ROTs by enabling the client to read from a slightly stale but stable snapshot defined by the read node and the client's cache. Then, we avoided the coordinator-based approach used in most stabilization protocols by restricting clients to their local read node, which tracks all partitions of its region, enabling one-round communication between client and server to perform a ROT. Like MongoDB [91], Contrarian [23], Wren [84], and PaRiS [85], the reference architecture uses HLCs to timestamp write operations. Together with the restriction described above, this enables ROTs to use constant metadata.

The comparison with the systems and approaches surveyed in the literature review also highlights the ability of the architecture to support partial replication on the server side at the cost of limiting clients to the partitions of their region. In contrast, PaRiS [85] enables clients to access data in partitions of other DCs, and SwiftCloud [98] only supports client-side replication.

The results of our validation also sustain the applicability of the architecture to read-heavy systems — read throughput increases almost linearly with the number of read nodes, and the average read latency is below 4ms in a saturated system (*cf.* section 8.4.2).

RQ5. How can this reference architecture guarantee CC above eventually consistent cloud storage services? What trade-offs arise from offering these guarantees?

This work integrates the literature's findings in a layered architecture that, akin to Bolt-on's approach, separates the causality concern from data replication and durability. This way, it supports eventually consistent cloud storage services provided that they follow a set of assumptions (section 6.1.3).

CC is enabled by the two top layers of the proposed architecture, which adapt the CANToR protocol [84, 85], where the snapshot of a ROT is the union of a causal snapshot installed by every partition in the client's local region and of his client-side cache. To enable CC atop an

ECDS, log versioning is also necessary to ensure read nodes see monotonically increasing versions of the data regardless of the order in which they are replicated.

Using AWS's cloud infrastructure, we demonstrate how the architecture can be realized in Amazon S3, which offers read-after-write consistency (chapter 7). By leveraging the methods exposed by Amazon S3's API and the encoding of the hybrid timestamps generated by the HLC, it is possible to guarantee that read nodes retrieve increasing versions of the log.

The empirical validation of our work suggests that the main trade-off of providing CC on top of existing storage services is staleness (*cf.* section 8.4.1). Enabling CC on top of Amazon S3 results in improved read latency and goodput but prejudices data visibility because writes are batched in a log and persisted periodically. This trade-off can be managed by adjusting the rate at which the log is persisted to the storage service.

RQ6. How can we realize this reference architecture to ensure *value semantics*?

The realized reference architecture enables *value semantics* through four main properties (*cf.* section 7.9). First, it features a determinism state machine behavior, where writes generate a new immutable version of a key, enabling the perception of each key's values in time. Secondly, by versioning logs and using hybrid timestamps, we can observe the value of a key in time and audit the state of a partition using physical time. Thirdly, through ROTs, users can perceive the system at a stable point in time. Finally, by using atomic operations, a key's state transparently transitions from one value to another.

9.3 Hypothesis Revisited

The research questions that drove this work aimed to validate the following hypothesis:

> *There exists a reference architecture that (1) manifests the ideal properties of geo-replicated causally consistent read-heavy systems, (2) upgrades the consistency guarantees of existing cloud storage services, and (3) enables value semantics, thereby facilitating auditing and enabling developers to reason about the system's state and data at a point in time.*

We now deconstruct this hypothesis and discuss how each premise has been addressed in this dissertation:

1) There exists a reference architecture that manifests the ideal properties of geo-replicated causally consistent read-heavy systems.

From **RQ1**, we concluded that to achieve better performance in read-heavy systems, ROTs must manifest the following properties: be non-blocking, take one round of on-path communication, and use constant metadata. As described in section 8.3, the proposed reference architecture ensures these three properties. First, to ensure the non-blocking property, the snapshot visible to a ROT is the union of a stable snapshot identified by the read node with the client's cache. Secondly, each read node is responsible for all partitions of its region, and clients are restricted to the items available in their local read node, ensuring that a ROT can be performed with a single on-path round. To ensure constant metadata, we used hybrid timestamps to version writes and, therefore, the `stableTime` in each server, the only additional metadata exchanged in a ROT, does not vary with the system's size.

2) The reference architecture upgrades the consistency guarantees of existing cloud storage services.

The proposed reference architecture takes a layered approach to CC, decoupling the causality concern, enabled by the two top layers, from replication and durability, which the data store provides. This way, it brings portability to CC, making it possible to upgrade the consistency guarantees of existing storage services, provided that they conform with a set of assumptions (section 6.1.3).

We further support this claim by realizing the reference architecture in a prototype system that uses Amazon S3 as the underlying storage service (chapter 7).

3) The reference architecture enables *value semantics*, thereby facilitating auditing and enabling developers to reason about the system's state and data at a point in time.

By adopting a deterministic state machine behavior where write nodes sequence writes in a log together with a hybrid timestamp, versioning the logs, providing ROTs that capture a stable snapshot of the data, and, finally, through atomic operations that transparently transition a key's state from one value to another, our reference architecture enables developers and users to perceive the system as a succession of immutable values, enabling *value semantics*. We experimentally demonstrated how providing *value semantics* makes it easier to audit the system by making it possible (1) to reason about the state of the system at a given point in time, (2) to observe the progression of values for a given key, and (3) to use physical time to audit the system.

9.4 Contributions

The work developed in this dissertation resulted in the following contributions to the state of the art of causally consistent distributed systems:

Systematic Literature Review on Causally Consistent Distributed Systems. We surveyed the state of the art of causally consistent distributed systems through a systematic literature

review (*cf.* chapter 3). A summary of this contribution has been submitted to the ACM Computing Surveys Journal and is currently under review.

Architectural Representation of Causally Consistent Distributed Systems. We extended the literature review by deriving the high-level architecture of four state-of-the-art distributed systems (*cf.* section 5.1).

Reference Architecture for Cloud-Native Causally Consistent Systems. We designed a reference architecture for cloud-native systems that enables CC atop existing cloud storage services (*cf.* chapter 6) and manifests the performance-optimal properties of ROTs.

Prototype Implementation and Validation. To verify the viability of the proposed architecture, we realized it in a prototype system that provides CC atop Amazon S3 and empirically validated it against its baseline storage service and in read-heavy workloads (*cf.* chapter 7 and section 8.4). To facilitate replicating and expanding over this work, we have prepared a replication package [30] that includes this implementation's source code and the instructions for replicating the validation experiments.

Value Semantics Realization in Read-Heavy Causally Consistent Systems. We provided progress towards defining the properties of *value semantics* in the scope of distributed data stores, realized those properties in our prototype implementation, and experimentally demonstrated how enabling *value semantics* provides increased auditability (*cf.* sections 2.3, 7.9 and 8.5).

9.5 Challenges

Throughout the development of this dissertation, we encountered several challenges.

One initial challenge lay in the literature's definition of performance-optimal ROTs. While this definition provides valuable insights into improving performance in read-heavy systems, it alone does not fully elucidate the reasoning behind Lu *et al.*'s [58] classification of some systems. Specifically, we felt the lack of an explanation regarding the classification of some systems concerning the metadata size and the number of off-path rounds in systems employing stabilization protocols. To address this, we performed our own analysis of each system.

During the development of the prototype system, we encountered a significant obstacle in testing the system locally since it was primarily designed to run in the cloud. To partially address this challenge, we used LocalStack for local development. However, LocalStack also presented its limitations, including the inability to test replication locally and to handle specific request rates and the need to configure certain calls to AWS SDK methods differently for local development. These limitations slowed the prototype testing process, as some features could only be assessed after deploying the system in AWS. Moreover, instances were terminated upon usage to minimize resource usage and costs, so, in each experiment, we had to set up the instances from scratch.

Although automating the testing process was not our research's focus, it could prove beneficial for future iterations or similar studies.

Defining the validation strategy posed a challenge due to the variations in programming languages, communication infrastructure, and cloud environments among different implementations. To conduct a meaningful comparison, it would have been necessary to re-implement the systems from scratch using the same code base, which was not feasible within the scope of this dissertation. Instead, our focus shifted towards assessing the impact of providing CC atop an existing storage service, leaving the comparison with state-of-the-art systems for future research.

Lastly, the concept of *value semantics* is not standardized in the scope of Distributed Systems. This led us to propose our definition based on our literature review (section 2.3).

9.6 Future Work

This work represents a significant step towards establishing a generic solution for implementing causally consistent read-heavy systems atop existing cloud storage services. Nevertheless, further refinements and extensions can be pursued in the following areas:

Support Non-Sticky Clients. The proposed architecture assumes that the end users will always contact the same read node for each session or desired scope of causality. Even though this "stickiness" assumption is standard in existing implementations of CC, in a real-world system, it may be necessary to bounce clients between read nodes because of failures or for load balancing. In this regard, one possible path forward is to upgrade the proposed reference architecture to support non-sticky clients. Provided that read nodes in the same region store the same set of partitions, this could be achieved by making clients store the last `stableTime` they know and send it in subsequent ROTs. In the worst-case scenario where the read node's `stableTime` is below the client's, it may synchronously fetch the latest log, update its `stableTime` and answer the client's request.

Support WOTs . Another possible extension of the proposed architecture and prototype would be to enable WOTs. To that end, a possible approach would be to make write nodes assign the same timestamp to all the transaction updates, ensuring that they are made visible together.

Furthermore, to transition from a prototype to a production-ready solution, and hence to ensure the practical applicability of the proposed solution in the industry, certain modifications would need to be considered:

Log Format. A potential approach would involve appending writes to a memory-mapped file to support durability while the log has not yet been persisted to Amazon S3. In this scenario, modifying the log format to register each write on a separate line would be advantageous. Additionally, this modification would facilitate parsing, enabling read nodes to easily keep track of the last version they have applied and help reduce storage overhead by avoiding the verbosity of JSON logs.

Failure Recovery. In this dissertation, we have discussed failure-recovery techniques for client and server failures, specifically focusing on checkpoint recovery (*cf.* section 6.1.6). However, due to time constraints, the prototype system does not currently address failure scenarios. Moving forward, it is essential to consider and implement failure-recovery mechanisms to enhance the system's resilience, possibly by integrating ideas from the research community.

For instance, one possibility would be to integrate SwiftCloud's [98] failure recovery strategy to ensure that updates only become visible when they become stable across a certain number of read nodes. This strategy would make it easier to support non-sticky clients without synchronous reads, although it would introduce additional communication and impact visibility.

Another possible direction forward involves enabling parallel recovery by maintaining a Distributed Hash Table (DHT) ring overlay where nodes store a fragment of their neighbor write nodes, ensuring that the write node's state can be efficiently reconstructed [52]

Synchronization in the absence of updates. In our proposed architecture, read nodes must continually observe new versions of each partition's log to advance the `stableTime` and make new versions visible. Consequently, write nodes must persist the log in the data store even in the absence of updates, resulting in duplicate logs. This duplication can potentially impact the ability to observe the partition's history since duplicates occupy space that could otherwise be used for storing stale log versions. Addressing these limitations and exploring alternative approaches, such as replacing the log's key when new updates cease to occur, may offer potential avenues for improvement in this work.

Another opportunity for improvement lies in clock synchronization. Our prototype system synchronizes each clock with other partitions using Amazon S3 to exchange clock values. While this approach avoids persisting the log when all partitions cease to receive updates, it requires additional requests to Amazon S3 and imposes extra storage costs. Alternatively, assuming physical clocks are loosely synchronized by a time synchronization protocol such as NTP [1], the current HLC value can be used to timestamp the log because, in the absence of incoming events, it still progresses with physical time.

Garbage Collection. Due to time constraints, the prototype system does not currently incorporate garbage collection of the logs stored in Amazon S3. However, implementing a garbage collection mechanism that leverages Amazon S3's programming interface is essential for transitioning from a prototype to a production-ready solution, as it ensures bounded storage.

Finally, it would be valuable to expand the verification and validation through the following approaches:

Extend the Empirical Validation. Following up on the validity threats identified in section 8.4.3, due to time and cost constraints, the empirical validation of the prototype system does not

encompass all possible load conditions, workloads, and testing scenarios. In this regard, possible extensions of this validation include assessing the prototype in skewed workloads, varying the number of keys per ROT, exploring different payload sizes, and incorporating additional regions.

Compare with State-of-the-Art Systems. The state-of-the-art review identifies 16 implementations of CC, which could serve as potential candidates for comparison with our prototype system. However, these implementations feature different programming languages and communication infrastructure and have been evaluated in a distinct cloud environment. To conduct a meaningful comparison, it would be necessary to reimplement the systems from scratch using the same code base as our prototype and evaluate them in a standardized environment. Unfortunately, due to time and cost constraints, this was not feasible within the scope of this dissertation. Nevertheless, performing such a comparison would be a valuable step forward in future research.

Realize the Reference Architecture with different Cloud Storage Services. Due to time constraints, the reference architecture was only realized in the scope of Amazon S3, a NoSQL key-value cloud storage service that provides read-after-write consistency. Therefore, its validity when implemented on top of other cloud storage services that offer pure EC or stronger semantics is yet to be tested in future research.

Realize the Reference Architecture in an Industry Scenario. Kevel's internal data distribution system is a CC+ key-value data store, which we were eager to compare against our prototype. We were unable to achieve this goal since the CC enforcement and client-cache management are part of the systems' logic (not implemented as a dedicated storage layer). Using our prototype as the storage layer within Kevel's system would be redundant and only introduce additional overhead. Nonetheless, we believe that the reference architecture can be readily integrated into existing production systems.

References

[1] Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305, March 1992.

[2] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[3] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing 1995 9:1*, 9:37–49, 3 1995.

[4] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 405–414, 2016.

[5] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 85–98, New York, NY, USA, 2013. Association for Computing Machinery.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[7] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, aug 1978.

[8] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 761–772, New York, NY, USA, 2013. Association for Computing Machinery.

[9] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8):776–787, apr 2012.

[10] Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. *Communications of the ACM*, 59(4):43–47, 2016.

[11] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, page 5, USA, 2006. USENIX Association.

[12] Kenneth P. Birman, Robert V. Renesse, and Robbert Van Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, Washington, DC, USA, 1994.

[13] John Lakos Bloomberg. Normative language to describe value copy semantics. 2007.

[14] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[15] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. {TAO}:{Facebook's} distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[16] Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. From session causality to causal consistency. In *PDP*, pages 152–158, 2004.

[17] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. 1993.

[18] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. Taobench: An end-to-end benchmark for social network workloads. *Proc. VLDB Endow.*, 15(9):1965–1977, jul 2022.

[19] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), aug 2013.

[20] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. D2stm: Dependable distributed software transactional memory. In *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 307–313, 2009.

[21] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[22] F. Cristian, H. Aghili, R. Strong, and D. Volev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.*, pages 431–, 1995.

[23] Diego Didona, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. Causal consistency and latency optimality: Friend or foe? *Proc. VLDB Endow.*, 11(11):1618–1632, jul 2018.

[24] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[25] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–13, New York, NY, USA, 2014. Association for Computing Machinery.

[26] Alan D. Fekete and Krithi Ramamritham. *Consistency Models for Replicated Data*, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[27] Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research - an initial survey. In *International Conference on Software Engineering and Knowledge Engineering*, 2010.

[28] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. 1988.

[29] Martin Fowler. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley, 2012.

[30] Diana Freitas. Towards Causal Consistency in Read-Heavy Cloud-Native Systems - Replication Package. https://doi.org/10.5281/zenodo.8091091, June 2023.

[31] Hector Garcia-Molina and Gio Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems (TODS)*, 7:209–234, 6 1982.

[32] GilbertSeth and LynchNancy. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33:51–59, 6 2002.

[33] J. A. Goguen. Semantics of computation. In Ernest Gene Manes, editor, *Category Theory Applied to Computation and Control*, pages 151–163, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.

[34] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, dec 1983.

[35] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[36] Rich Hickey. Are we there yet. https://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey/, 2009. Presented at the JVM Language Summit 2009.

[37] Rich Hickey. The value of values with rich hickey. https://www.youtube.com/watch?v=-6BsiVyC1kM, 2012. Presented at the JAX conference 2012.

[38] Rich Hickey. A history of clojure. 4(HOPL), jun 2020.

[39] Ta-Yuan Hsu and Ajay D. Kshemkalyani. Cadrop: Cost optimized convergent causal consistency in social network systems. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 426–435, 2021.

[40] Diptanshu Kakwani and Rupesh Nasre. Orion: Time estimated causally consistent key-value store. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[41] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 654–663, New York, NY, USA, 1997. Association for Computing Machinery.

[42] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193)*, pages 68–78, 1998.

[43] Martin Kleppmann. *Designing data-intensive applications*. O'Reilly Media, Sebastopol, CA, March 2017.

[44] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems*, pages 17–32, Cham, 2014. Springer International Publishing.

[45] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, nov 1992.

[46] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95–114, 1978.

[47] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978.

[48] Leslie Lamport. The part-time parliament. 16(2):133–169, may 1998.

[49] Leslie Lamport. Paxos made simple. 2001.

[50] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. OSDI'12, page 265–278, USA, 2012. USENIX Association.

[51] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[52] Pinchao Liu, Hailu Xu, Dilma Da Silva, Qingyang Wang, Sarker Tanzir Ahmed, and Liting Hu. Fp4s: Fragment-based parallel state recovery for stateful stream applications. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1102–1111, 2020.

[53] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David 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*, SOSP '11, page 401–416, New York, NY, USA, 2011. Association for Computing Machinery.

[54] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for Low-Latency Geo-Replicated storage. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 313–328, Lombard, IL, April 2013. USENIX Association.

[55] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual consistency: Stronger properties for low-latency geo-replicated storage. *Queue*, 12(3):30–45, mar 2014.

[56] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW theorem and Latency-Optimal Read-Only transactions. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 135–150, Savannah, GA, November 2016. USENIX Association.

[57] Haonan Lu and Siddhartha Sen. Performance-optimal read-only transactions (extended version). 2022.

[58] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. Performance-Optimal Read-Only transactions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 333–349. USENIX Association, November 2020.

[59] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at facebook. SOSP '15, page 295–310, New York, NY, USA, 2015. Association for Computing Machinery.

[60] Taras Lykhenko, Rafael Soares, and Luis Rodrigues. Faastcc: Efficient transactional causal consistency for serverless computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 159–171, New York, NY, USA, 2021. Association for Computing Machinery.

[61] B. J. MacLennan. Values and objects in programming languages. *ACM SIGPLAN Notices*, 17(12):70–79, December 1982.

[62] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, et al. Consistency, availability, and convergence. *University of Texas at Austin Tech Report*, 11:158, 2011.

[63] Tariq Mahmood, Shankaranarayanan Puzhavakath Narayanan, Sanjay Rao, T. N. Vijaykumar, and Mithuna Thottethodi. Karma: Cost-effective geo-replicated cloud storage with dynamic enforcement of causal consistency. *IEEE Transactions on Cloud Computing*, 9(1):197–211, 2021.

[64] Friedemann Mattern et al. *Virtual time and global states of distributed systems*. Univ., Department of Computer Science, 1988.

[65] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 453–468, USA, 2017. USENIX Association.

[66] John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2002.

[67] A. R. Nasibullin and B. A. Novikov. Replication in distributed systems: Models, methods, and protocols. *Programming and Computer Software*, 46(5):341–350, September 2020.

[68] Khiem Ngo, Haonan Lu, and Wyatt Lloyd. K2: Reading quickly from storage across many datacenters. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 199–211, 2021.

[69] Shadi A. Noghabi, Sriram Subramanian, Priyesh Narayanan, Sivabalan Narayanan, Gopalakrishna Holla, Mammad Zadeh, Tianwei Li, Indranil Gupta, and Roy H. Campbell. Ambry: Linkedin's scalable geo-distributed object store. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 253–265, New York, NY, USA, 2016. Association for Computing Machinery.

[70] Marlies Noordzij, Lotty Hooft, Friedo W. Dekker, Carmine Zoccali, and Kitty J. Jager. Systematic reviews and meta-analysis: when they are useful and when to be careful. *Kidney International*, 76(11):1130–1136, 2009.

[71] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, 1988.

[72] Chitu Okoli and Kira Schabram. A guide to conducting a systematic literature review of information systems research. *Research Methods & Methodology in Accounting eJournal*, 2010.

[73] Diego Ongaro and John Ousterhout. Raft. USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[74] Douglas Stott Parker, Gerald J. Popek, Gerard Rudisin, Alley Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David A. Edwards, Stephen Kiser, and Charles S. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9:240–247, 1983.

[75] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. *SIGOPS Oper. Syst. Rev.*, 31(5):288–301, oct 1997.

[76] Nicolas Poggi. *Microbenchmark*, pages 1143–1152. Springer International Publishing, Cham, 2019.

[77] Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Dotted version vectors: Logical clocks for optimistic replication. *arXiv preprint arXiv:1011.5808*, 2010.

[78] M. Raynal and M. Singhal. Logical time: capturing causality in distributed systems. *Computer*, 29(2):49–56, 1996.

[79] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990.

[80] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[81] Justin Sheehy. There is no now: Problems with simultaneity in distributed systems. *Queue*, 13(3):20–27, mar 2015.

[82] Hannah Snyder. Literature review as a research methodology: An overview and guidelines. *Journal of Business Research*, 104:333–339, 2019.

[83] Evjola Spaho, Leonard Barolli, and Fatos Xhafa. Data replication strategies in p2p systems: A survey. In *2014 17th International Conference on Network-Based Information Systems*, pages 302–309, 2014.

[84] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Wren: Nonblocking reads in a partitioned transactional causally consistent data store. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2018.

[85] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Paris: Causally consistent transactions with non-blocking reads and partial replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 304–316, 2019.

[86] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Optimistic causal consistency for geo-replicated key-value stores. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):527–542, 2021.

[87] Alexander Stepanov and Paul McJones. *Elements of Programming*. Semigroup Press, June 2019.

[88] Yu Tang, Hailong Sun, Xu Wang, and Xudong Liu. Achieving convergent causal consistency and high availability for cloud storage. *Future Generation Computer Systems*, 74:20–31, 2017.

[89] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, 1994.

[90] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. SOSP '13, page 309–324, New York, NY, USA, 2013. Association for Computing Machinery.

[91] Misha Tyulenev, Andy Schwerin, Asya Kamsky, Randolph Tan, Alyson Cabral, and Jack Mulrow. Implementation of cluster-wide logical clock and causal consistency in mongodb. SIGMOD '19, page 636–650, New York, NY, USA, 2019. Association for Computing Machinery.

[92] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[93] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, 2004.

[94] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 3.03 edition, 2017.

[95] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, jan 2009.

[96] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 464–474, 2000.

[97] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Transactional causal consistency for serverless computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 83–97, New York, NY, USA, 2020. Association for Computing Machinery.

[98] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, page 75–87, New York, NY, USA, 2015. Association for Computing Machinery.

[99] M.V. Zelkowitz and D.R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, 1998.

[100] Xi Zhang, Alma Riska, and Erik Riedel. Characterization of the e-commerce storage subsystem workload. In *2008 Fifth International Conference on Quantitative Evaluation of Systems*, pages 297–306, 2008.

[101] Ying Zhang. Revisiting time, clocks, and synchronization, 2021.

Appendix A

Literature review results

System	First Published	Data Source	Review
COPS [53]	2011	ACM Digital Library	Section 3.2.1.1
Eiger [54]	2013	ACM Digital Library	Section 3.2.1.2
ChainReaction [5]	2013	ACM Digital Library	Section 3.2.1.3
Orbe [24]	2013	ACM Digital Library	Section 3.2.1.4
Bolt-on [8]	2013	ACM Digital Library	Section 3.3.1
GentleRain [25]	2014	ACM Digital Library	Section 3.2.1.5
SwiftCloud [98]	2014	ACM Digital Library	Section 3.2.1.6
COPS-SNOW [56]	2016	ACM Digital Library	Section 3.2.1.8
Cure [4]	2016	IEEE Xplore	Section 3.2.1.7
CoCaCo [88]	2017	ACM Digital Library	✗
Occult [65]	2017	ACM Digital Library	Section 3.2.1.9
POCC [86]	2017	IEEE Xplore	Section 3.2.1.10
Karma [63]	2018	IEEE Xplore	✗
Wren [84]	2018	IEEE Xplore	Section 3.2.1.11
Contrarian [23]	2018	ACM Digital Library	Section 3.2.1.12
PaRiS [85]	2019	IEEE Xplore	Section 3.2.1.13
MongoDB [91]	2019	ACM Digital Library	Section 3.3.2
Orion [40]	2020	ACM Digital Library	✗
PORT [58]	2020	ACM Digital Library	Section 3.2.1.14
HYDROCACHE [97]	2020	ACM Digital Library	✗
CaDRoP [39]	2021	IEEE Xplore	✗
K2 [68]	2021	IEEE Xplore	✗
FaaSTCC [60]	2021	ACM Digital Library	✗

Table A.1: Results of the systematic review.

Appendix B

Prototype Communication Structures

This appendix exemplifies the message interchange and log format used in our prototype system.

B.1 Protocol Buffers Messages and Services

This section depicts the message interchange format used in the prototype and the gRPC services provided by server nodes. All messages and services are specified using Protocol Buffers.

Listing B.1 exemplifies the message format used in ROTs and the service provided by read nodes. Listing B.2 presents the format used for write and atomic write messages and the service provided by write nodes. Finally, listing B.3 presents the messages used by write nodes to obtain the read nodes' `stableTime` and the service exposed by read nodes to provide this value.

B.1.1 ROTs Messages and Services

```
message KeyVersion {
    string timestamp = 1;
    bytes value = 2;
}

message ROTRequest {
    repeated string keys = 1;
}

message ROTResponse {
    map<string, KeyVersion> versions = 1;
    string stableTime = 2;
    int64 id = 3;
}

service ROTService {
    rpc rot (ROTRequest) returns (ROTResponse);
}
```

Listing B.1: Prototype Communication Structures — ROT Protocol Buffers Messages and Services. ROT requests include the keys to read. ROT responses include a version of the keys, the stable time, and the transaction's id. The transaction id and the timestamp of each version were only included for testing.

```
message WriteRequest {
    string key = 1;
    bytes value = 2;
    string lastWriteTimestamp = 3;
    optional string expectedVersion = 4;
    optional bytes expectedValue = 5;
}

message WriteResponse {
    string writeTimestamp = 1;
    optional string currentVersion = 2;
}

service WriteService {
    rpc write (WriteRequest) returns (WriteResponse);
    rpc atomicWrite (WriteRequest) returns (WriteResponse);
}
```

Listing B.2: Prototype Communication Structures — Write Protocol Buffers Messages and Services. Write requests include the key and value to write and the timestamp of the client's last write. Atomic writes may also include the expected value or timestamp. The response includes the timestamp of the new write and, in case of a failed atomic write, the timestamp of the current version of the key.

```
message StableTimeRequest {}

message StableTimeResponse {
    string stableTime = 1;
}

service StableTimeService {
    rpc stableTime (StableTimeRequest) returns (StableTimeResponse);
}
```

Listing B.3: Prototype Communication Structures — Checkpointing Protocol Buffers Messages and Services. The response message includes the `stableTime` of the read node.

B.2 Log Format

This section includes examples of the JSON log used in the prototype system. Listing B.4 presents an example of a log and listing B.5 exemplifies the effect of checkpointing.

```
{
  "state": [
    {
      "versions": [
        {
          "value": "1",
          "timestamp": "00000001687424433835-00000000000000000000"
        },
        {
          "value": "2",
          "timestamp": "00000001687424433835-00000000000000000001"
        }
      ],
      "key": "x"
    },
    {
      "versions": [
        {
          "value": "8",
          "timestamp": "00000001687424433835-00000000000000000002"
        },
        {
          "value": "9",
          "timestamp": "00000001687424453835-00000000000000000001"
        }
      ],
      "key": "y"
    },
    {
      "versions": [
        {
          "value": "14",
          "timestamp": "00000001687424453835-00000000000000000000"
        }
      ],
      "key": "z"
    }
  ]
}
```

Listing B.4: Prototype Communication Structures — Log format. For each key, an array orders the versions of that key, each comprising its value and hybrid timestamp.

```
{
  "state": [
    {
      "versions": [
        {
          "value": "2",
          "timestamp": "00000001687424433835-00000000000000000001"
        }
      ],
      "key": "x"
    },
    {
      "versions": [
        {
          "value": "9",
          "timestamp": "00000001687424453835-00000000000000000001"
        }
      ],
      "key": "y"
    },
    {
      "versions": [
        {
          "value": "14",
          "timestamp": "00000001687424453835-00000000000000000000"
        }
      ],
      "key": "z"
    }
  ]
}
```

Listing B.5: Prototype Communication Structures — Checkpointing. The result of performing a checkpoint on the log of listing B.4 using timestamp "00000001687424453835-00000000000000000001". The stale versions of keys x and y are discarded because they are already stable across read nodes. The last stable version of each key is preserved.