TÉCNICO LISBOA

Causality Tracking Trade-offs for Distributed Storage

Hugo Rafael Silva Guerreiro

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Luís Eduardo Teixeira Rodrigues
Prof. Nuno Manuel Ribeiro Preguiça

Examination Committee

Chairperson: Prof. Name of the Chairperson
Supervisor: Prof. Luís Eduardo Teixeira Rodrigues
Member of the Committee: Prof. Carlos Miguel Ferraz Baquero Moreno

October 2020

Acknowledgments

I would like to begin by thanking my advisors, Professor Luís Rodrigues and Nuno Preguiça for taking me as their student and for their patience and guidance throughout this year, which was crucial for the completion of this thesis. I would also like to thank Nivia Quental for her insights and kind words throughout this year.

To my great friends, João Sousa, Lucas Rafael, Marco Silva, Matilde Ramos, and Rodrigo Oliveira thank you not only for your support during these last few months but also for the journey we shared during these 5 years. Your friendship made the hardest days easier and helped me grow as a person.

I want to thank my mother Cristina Guerreiro and my father Fernando Silva for always believing in me and for making me push my limits. Thank you for the sacrifices you have made so that I could always pursue what I am passionate about. I cannot thank you enough.

To each and every one of you – Thank you.

This work was partially supported by the FCT via project NG-STORAGE, with reference PTDC/CCI-INF/32038/2017, and by projects INESC-ID (ref. UIDB/ 50021/ 2020) and NOVA LINCS (ref. UIDB/ 04516/ 2020).

Abstract

After the seminal paper by L. Lamport, which introduced (scalar) logical clocks, several other data structures for keeping track of causality in distributed systems have been proposed, including vector and matrix clocks. These are able to capture causal dependencies with more detail but, unfortunately, also consume a substantially larger amount of network bandwidth and storage space than Lamport clocks. This raises the question of whether the benefits of these more complex structures are worth their cost. We address this question in the context of partially replicated systems. We show that for some workloads the use of more expensive clocks does bring significant benefits and that for other workloads no visible benefits can be observed. In this thesis, we provide a characterization of the scenarios where each type of clock is more beneficial, helping designers to develop more efficient distributed storage systems.

Keywords

Distributed Storage; Partial Replication; Causal Consistency; Causal Order; Logical Clocks

Resumo

Após a introdução de relógios lógicos (escalares) por L. Lamport no seu trabalho seminal, várias outras estruturas de dados utilizadas para rastrear a causalidade em sistemas distribuídos foram propostas. Entre elas incluem-se os relógios vetoriais e matriciais que têm a capacidade de capturar dependências causais com maior precisão. Infelizmente, comparando com os relógios de Lamport, estas estruturas consomem uma quantidade de largura de banda de rede e espaço de armazenamento substancialmente maior. Este facto levanta a questão de se os benefícios que elas trazem compensam o seu custo. Abordamos esta questão no contexto de sistemas parcialmente replicados e mostramos que, para alguns perfis de carga, o uso de relógios mais caros traz benefícios significativos e que, para outros, nenhum benefício pode ser observado. Nesta dissertação fornecemos uma caracterização dos cenários onde cada tipo de relógio é mais benéfico, ajudando os programadores a desenvolver sistemas de armazenamento distribuído mais eficientes.

Palavras Chave

Armazenamento Distribuído; Replicação Parcial; Coerência Causal; Ordem causal; Relógios Lógicos

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

1L	One Lamport clock per system
1M	One Matrix clock per system
1V	One Vector clock per system
CC	Causal Consistency
CMO	Consistency Maintenance Overhead
CRDT	Conflict-free Replicated Data Types
FIFO	First-In First-Out
GRA	Update Generation Rate Asymmetry
OPR	Object ownership to objects in causal Past Ratio
TCP	Transmission Control Protocol
kL	one Lamport clock per object
kV	one Vector clock per object

1

Introduction

Contents

1.1 Motivation

The notion of causality is a fundamental concept in the context of distributed systems, and the ability to capture cause-effect relations accurately is a requirement of many distributed tasks, such as taking global snapshots [1], implementing distributed mutual exclusion [2], or maintaining the consistency of data [3]. After the seminal paper by L. Lamport [4], that introduced (scalar) logical clocks, several other data structures for keeping track of causality in distributed systems have been proposed, including vector [5, 6] and matrix clocks [7], among others. Each of these mechanisms offers a different trade-off between the space complexity and the ability to track causal dependencies accurately. Note that Lamport clocks consist of a single scalar (the space complexity is $\mathcal{O}(1)$), vector clocks typically require an entry per node in the system (the space complexity is $\mathcal{O}(N)$), and matrix clocks, as their name suggests, have a space complexity of $\mathcal{O}(N^2)$. Thus, despite their potential advantages, vector and matrix clocks consume a substantially larger amount of network bandwidth and storage space. This raises the question of whether, in practice, the benefits of these more complex structures are worth their cost. From the abstract point of view, it is clear that larger clocks can capture causality more accurately and reduce the remote visibility latency. Assume that all updates are timestamped with some form of logical clocks to keep track of causality. Let $t(u)$ be the timestamp assigned to update u. It is well known that, with Lamport clocks $u_x \rightarrow u_y \implies t(u_y) > t(u_x)$ but that $t(u_y) > t(u_x) \not\Longrightarrow u_x \rightarrow u_y$. This means that, when using Lamport clocks, a node that has received u_y and not u_x, cannot immediately assess if it is safe to deliver u_y. We call this artifact of Lamport clocks a *false dependency*. False dependencies have a negative impact on the remote visibility latency, by forcing updates to wait for other updates that may not be in their causal past. Vector clocks suffer less from this problem. However, to completely avoid false dependencies with vector clocks, updates need to be timestamped with not only one vector clock, but with a set of vector clocks, one vector clock for each data item. When the number of items is larger than the number of nodes in the system (arguably, the most common case) accurate clocks degenerate in a matrix clock. For any reasonably sized system, this has a significant overhead, both in terms of storage space and in terms of bandwidth utilization. Thus, in practice, the costs of maintaining accurate clocks can outweigh the potential advantages they may bring in terms of reduced visibility latency.

We study the trade-off between the space complexity of the vector clocks and the remote visibility latency in partially replicated storage systems under different workloads and different replica deployments. We show that for some workloads/deployments the use of more expensive clocks does bring significant benefits and that for others no visible benefits can be observed.

Unfortunately, there are many factors that affect the performance of a certain type of clocks, including: how many replicas of each object are kept and where these replicas are located, the read/write ratio of the workload, the access frequency to each object, the time between consecutive operations executed by clients, among others. This makes the task of identifying the best clock for a given scenario very

hard.

To solve this puzzle, we introduce two novel features that capture relevant patterns for the performance of causality tracking mechanisms: the *Update Generation Rate Asymmetry* (GRA), defined as the ratio between the fastest and slowest average update frequency for all nodes in the system, and the *Object Ownership to Objects in Causal Past Ratio* (OPR), defined has the ratio between the number of objects replicated at a node and the objects with updates in the causal frontier of an update. With these new features, the thesis provides a characterization of the scenarios where each type of clock is more beneficial, helping system designers to develop more efficient distributed storage systems.

1.2 Research history

This work was performed in the context of a research project, named NG-STORAGE, that aims at finding efficient mechanisms to store data on the network edge. A key aspect of this project is to find the right amount of metadata that needs to be stored and exchanged to ensure that updates are applied in causal order at all edge nodes. My work aims at finding the best metadata as a function of the workload profile. The work described in the thesis has been partially published in a paper, namely:

”H. Guerreiro, L. Rodrigues, N. Preguiça and N. Quental. Causality Tracking Trade-offs for Distributed Storage. In IEEE NCA 2020” [8]

This work was partially supported by the FCT via project NG-STORAGE, with reference PTDC/CCI-INF/32038/2017, and by projects INESC-ID (ref. UIDB/ 50021/ 2020) and NOVA LINCS (ref. UIDB/ 04516/ 2020).

1.3 Contributions

This thesis studies existing logical clocks data structures and identifies a set of features that help to understand the performance of the different mechanisms. The main resulting contributions of this dissertation are:

- The proposal of two novel metrics, the *Update Generation Rate Asymmetry* (GRA) and the *Object Ownership to Objects in Causal Past Ratio* (OPR), that capture key aspects of the workload for the purpose of selecting the best metadata to keep track of causality.

- A decision tree to help system designers choose the best logical clock based on the values of the two metrics above.

1.4 Results

This thesis achieved the following results:

- A common framework to support the implementation and comparison of different causal order algorithms.
- An implementation of the selected algorithms using the Peersim simulator.
- An experimental evaluation and comparison of different algorithms to keep track of causal order in face of multiple workloads with different characteristics.

1.5 Organization of the Document

This thesis is organized as follows:

- Chapter 2 presents important definitions regarding causality in distributed systems (namely, happened-before and causal consistency) and related work that stems from said definitions.
- Chapter 3 introduces the different logical clocks considered in the thesis and how they can be used to enforce causal consistency.
- Chapter 4 describes some of the implementation details of the prototypes used to test the different logical clocks and algorithms.
- Chapter 5 presents an empirical study of different algorithms under different workloads.
- Chapter 6 defines a set of guidelines to help decide between the different algorithms.
- Chapter 7 concludes the thesis and unveils some possible lines of work for future work.

2

Related Work

Contents

In this chapter, we discuss the concept of time in the context of distributed systems, the different mechanisms we use to track it, and how we can use them to establish an order to events. We focus on a specific ordering of events named causal order that is used to build the causal consistency model and we finish by addressing some known problems of ensuring causal consistency.

As such, Section 2.1 describes time in distributed systems; Section 2.2 presents causal order, the definition of happened-before and the different mechanisms to track causal dependencies; Section 2.3 shows how causal order can be adapted to construct a consistency model, namely causal consistency; Finally, Section 2.4 discusses some of the trade-offs that are present when providing causal consistency.

2.1 Time in Distributed Systems

Time is an essential concept when reasoning about how events are ordered in a system; it ties well with our way of thinking and numerous applications rely on accurate time tracking to function correctly. Being able to assign a timestamp to an event accurately is of paramount importance. For example, in an application that manages bank transactions, it is of extreme importance that a withdrawal operation doesn't happen in the system before a previous money deposit (i.e., it is assigned a lower timestamp).

Various problems in distributed systems rely on the usage of clocks to be solved. The problem of data consistency is especially interesting in this context since it relies on establishing a correct order of operations (in fact, the banking example can be modeled as a data consistency problem). This order is typically based on the time each operation was first observed by the system. Since there can exist multiple observers (i.e., multiple processes), defining an order of events is not trivial and depends on what has been observed by each process.

A simple way to address this problem would be to rely on the physical clocks that are present in each machine to assign timestamps to events. However, these devices are not perfect and their readings are prone to diverge from one another. This problem can result in events being timestamped incorrectly and, therefore, result in inconsistent data versions at different processes. To mitigate this issue it is required that clocks are frequently synchronized through the usage of an external source of highly accurate time (e.g. NTP, Coordinated Universal Time UTC [9]). However, the quality of the synchronization attainable through these algorithms may vary significantly and depends on factors such as the network load. This fluctuation leads to unbounded uncertainty intervals that can vary from a few to hundreds of milliseconds. To guarantee clear and well-defined uncertainty intervals, one would require specialized hardware devices such as atomic clocks, which are expensive and usually not available or easily accessible.

Intuitively, events are ordered from the point of reference of every process, which means they need to use local information to assign timestamps to events (i.e. their local clock). For the reasons mentioned above and as Lamport pointed out in his seminal work [4], since it is not easy to synchronize physical

clocks in a distributed system, we cannot rely on this physical time to determine the order of any random pair of events without having a global picture of the system. Timestamping events according to the cause-effect relationship that they present when using physical time (i.e, physical causality) is proven useful, which led to the proposal of a similar type of ordering usually named *causal ordering* or *potential causal ordering*. This type of ordering introduced what is known as logical time, which can be tracked using data-structures named logical clocks. In the next section, we present this type of partial ordering in more detail.

2.2 Causal Order in Message Passing Systems

The notion of cause-effect is intrinsic to the way we perceive the world. For example, it is intuitive that if I am to pick up a rock from the ground and drop it, the rock will fall and hit the ground again. This is a case of a direct *cause-effect* relationship: the fact that I dropped the rock, resulted in it falling. However, other events may have influenced the rock falling. For instance, millions of years of erosion eventually resulted in the small rock being on the ground to be picked up in the first place. Since we cannot tell for sure if erosion was the cause of the rock falling, we assume that it *potentially caused* it. In distributed systems, this concept was formalized by L.Lamport which introduced the *happened-before* relationship as the basis for *potential causal ordering*. The happened-before relationship is based on two intuitive aspects:

- first, if two events occurred in the same process then they occurred in the order in which said process observed them.
- second, when a process sends a message, it is clear that the event of sending said message cannot happen after the message has been received by other processes.

Happened-before: More formally, an event e_1 is said to have *happened-before* an event e_2 ($e_1 \rightsquigarrow e_2$) iff:

(i) If e_1 and e_2 are events that happened in the same process and e_1 occurred before e_2 then $e_1 \rightsquigarrow e_2$;

(ii) If e_1 is the event that corresponds to a message being sent by a process and e_2 is an event that corresponds to that message being received by some other process, then $e_1 \rightsquigarrow e_2$;

(iii) If $e_1 \rightsquigarrow e_2$ and $e_2 \rightsquigarrow e_3$ then $e_1 \rightsquigarrow e_3$.

To better understand this definition, consider the example in Figure 2.1. Consider the interactions between the three processes $p1$, $p2$ and $p3$. In this scenario, it is clear that a happens-before b because they happen in the same process, therefore we can say that $a \rightsquigarrow b$ (i). Event b generates message $m1$ which upon arrival at process $p3$ also generates event c. Given this fact, it is immediate that event c

happened-before event b, therefore we say that $b \rightsquigarrow c$ (ii). Since $a \rightsquigarrow b$ and $b \rightsquigarrow c$ we can say that $a \rightsquigarrow c$ (iii). For similar reasons, $a \rightsquigarrow f$. Notice that we cannot determine whether $a \rightsquigarrow e$ or $e \rightsquigarrow a$ because we cannot find any causal link between the two events. In this scenario we say that events a and e are *concurrent* and we represent it as $a \| e$.

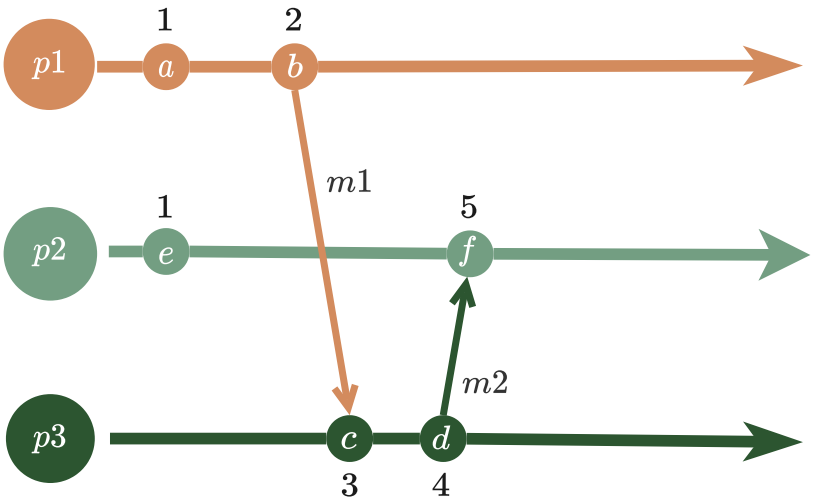


Figure 2.1: Example of the evolution of Lamport clocks

The design of distributed applications can be simplified if there is a communication layer that delivers messages to processes in an order that respects the *happened-before* relation [10, 11]. Mechanisms to enforce causal order have been widely studied in the literature [11]. These techniques typically rely on data structures called logical clocks to operate. Next, we present some of the most widely used logical clocks.

2.2.1 Lamport Clocks

Lamport clocks are essentially software scalars that increase monotonically and do not share any dependencies with real physical devices (unlike physical clocks). For the sake of exposition and to understand how Lamport clocks work, let's consider the original definition in the context of message-passing systems. A Lamport clock L_i associated with an arbitrary process p_i is defined as follows:

(i) Initially $L_i = 0$.

(ii) Before any event happens at p_i, L_i is incremented: $L_i := L_i + 1$, and everytime process p_i sends a message, its value L_i is *piggybacked* in the message.

(iii) On the receiving site, process p_j computes $L_j := max(L_j + 1, L_i)$.

The behavior of Lamport clocks is depicted in the example of Figure 2.1. The use of Lamport clocks is sufficient to enforce causal ordering. In fact, given two events e_1 and e_2 we have $e_1 \rightsquigarrow e_2 \Rightarrow L(e_1) < L(e_2)$; The contrary, however, is not true ($L(e_1) < L(e_2) \not\Rightarrow e_1 \rightsquigarrow e_2$). This means that Lamport clocks are not sufficient to determine whether two events e_1 and e_2 are concurrent or not. For example, in the scenario of Figure 2.1 it is not possible to determine if $a \| e$.

Notice that there is a difference between what we use to enforce causal ordering and how we actually enforce it. The definition of a Lamport clock presented in this section only specifies that they can be used as a mechanism to enforce causal ordering, setting aside how to achieve it. Therefore, in Section 3.1 we expose how we can use these mechanisms to achieve causal multicast in a partial replication context.

2.2.2 Vector Clocks

To overcome the aforementioned limitation, vector clocks were introduced [6, 12]. A vector clock joins multiple logical clocks, typically one for each process in the system - e.g., if the system has N processes, the vector clock will contain N Lamport clocks. Similarly to Lamport clocks, each process p_i keeps a vector clock V_i which is updated every time an event happens, and the vector clock is piggybacked in every message sent by a process. The rules for updating a vector clock in message-passing systems are:

(i) Each entry in the vector V_i is initialized to 0

(ii) Each time an event occurs at process p_i , its vector clock is incremented $V_i[i] = V_i[i] + 1$

(iii) When process p_i receives a message from another process p_j, the value of the vector clock is set to the pairwise maximum of each entry in both clocks: $V_i[k] := max(V_i[k], V_j[k])$, for $k = 1, 2, ..., N$.

In Figure 2.2 we depict the same interactions as in Figure 2.1 but change the timestamps to show the behavior of vector clocks instead. Unlike logical clocks, this mechanism ensures that for two events e_1 and e_2 the following is true: $e_1 \rightsquigarrow e_2 \Rightarrow L(e_1) < L(e_2)$; $L(e_1) < L(e_2) \Rightarrow e_1 \rightsquigarrow e_2$. This property makes it possible to compare two vectors in such a way that we can not only identify whether one event happened before other, but also if two events are concurrent.

Although *vector clocks* are a better approximation of causality than logical clocks, they have the disadvantage of growing linearly in size with the number N of processes in the system. As N grows large, so does the penalty incurred in terms of bandwidth and storage. In later sections, we study these trade-offs in more detail.

Matrix clocks: Several other causality tracking mechanisms and optimizations have been proposed in the literature [5, 13–19]. Among all other data structures we give special attention to *Matrix clocks* [7]. *Matrix clocks* extend the concept of vector clocks in another dimension. They allow us to capture causal

relationships at the granularity of a single network link by keeping one vector clock for each process in the system.

Hybrid Logical Clock: Another particularly interesting data structure is the hybrid logical clock [19]. Which mixes both physical time and logical time. The hybrid logical clock timestamp involves two parts: i) the physical part which contains a physical clock and ii) the logical part which contains a logical clock, typically a single scalar.

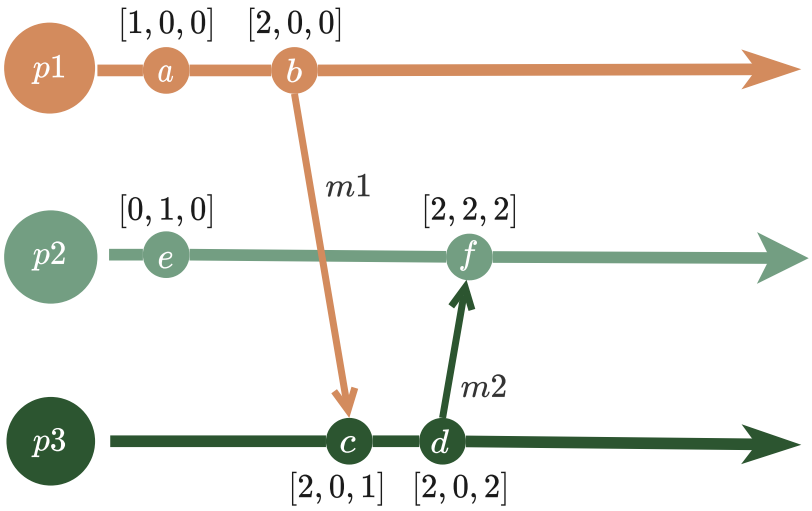


Figure 2.2: Example of the evolution of vector clocks

2.3 Causal Consistency in Shared Memory

The *happened-before* relation was originally defined in the context of message-passing systems, but it also applies to shared memory systems where processes interact by reading and writing on objects. In this case, we can define the cause-effect relations as follows:

(i) **Thread of Execution**. If a and b are two operations executed by the same thread of execution (for instance, by the same client), then $a \rightsquigarrow b$ if a is executed before b.

(ii) **Reads From**. If a is an update operation and b is a read operation that reads the value written by a, then $a \rightsquigarrow b$.

(iii) **Transitivity**. If $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

More specifically, let $w(o_a)$ and $w(o_b)$ be two write operations on, o_a and o_b. Let $r(o_a)$ and $r(o_b)$ be two read operations by the same client, where $r(o_a)$ is executed before $r(o_b)$ and where $r(o_a)$ returns

the value written by $w(o_a)$ and $r(o_b)$ returns the value written by $w(o_b)$. This execution satisfies *Causal Consistency* (Causal Consistency (CC)) [3] if there is no write $w'(o_b)$ such that $w(o_b) \rightsquigarrow w'(o_b) \rightsquigarrow w(o_a)$.

Causal consistency is extremely relevant in the context of highly-available distributed storage as it has been proven to be the strongest consistency model that can provide availability in face of transient network partitions [20, 21]. Stronger consistency models, such as serializability [22] or linearizability [23], require processes to reach consensus and to establish a total order among concurrent operations, therefore, being prone to blocking [24]. One of the most common strategies to achieve causal consistency in distributed and replicated storage systems is to ensure that updates performed at remote nodes are applied to any replica in causal order.

Notice that by allowing concurrent updates to execute at different orders in different processes, the states may diverge. A slightly stronger model named *Causal+* [25] has been proposed, where the additional condition of eventual and independent conflict resolution of concurrent updates needs to be ensured. This condition is such that the states of every process will eventually reach the same value. Conflict-free replicated data types (Conflict-free Replicated Data Types (CRDT)) [26] are a known way to implement causal+ consistency.

2.3.1 Causally Consistent Partially Replicated Systems

The use of data replication is almost unavoidable in modern distributed systems. First, data replication is a fundamental technique to provide fault-tolerance, a key requirement in most systems. Second, when the applications have clients that reach the system from different geographic locations, only replication can ensure that these clients access data with low latency. Having all existing data in the system present at all replicas increases not only its redundancy but also the storage and bandwidth costs associated with having to maintain said data. Partial replication is a technique where each data object is potentially only replicated in a subset of all replicas. This approach manages to reduce the mentioned costs.

A major part of the existing causally consistent systems assumes that data is fully replicated. Having data fully replicated allows designers to make the assumption that all replicas eventually receive all updates, which in turn allows metadata to be more compressed. If we consider partial replication, this cannot be assumed.

We now raise awareness of the difficulties involved with achieving causal consistency in a partial replication setting and present an overview of some systems that were able to achieve it.

2.3.1.A Causal Consistency vs Partial Replication

Achieving causal consistency under full replication has been widely studied [25, 27–35]. These systems implement protocols that are usually efficient when compared with protocols that implement strong consistency models. When transposing to partial replication, using the same techniques may not work;

those that do, either require extra metadata to be maintained or further coordination between replicas to obtain said metadata. One way or the other, both cases have scalability problems.

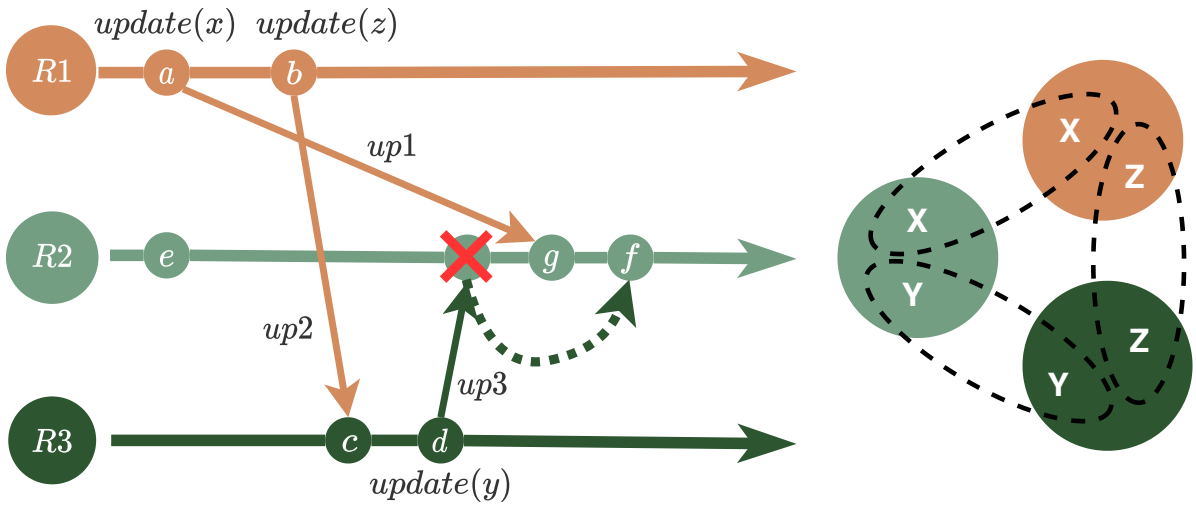


Figure 2.3: Events leading to a break in causality in a genuine partial replication scenario

In an ideal scenario, each partial replica would only maintain causal information regarding data objects that are currently being replicated. Attaining *genuine partial replication* [36] without breaking causality is difficult and might even be impossible without further coordination between replicas or defining a replica communication topology (e.g., Saturn [37]). To give a sense of why causally consistent genuine partial replication is hard to accomplish, consider the scenario displayed in Figure 2.3. In this example, we consider a distributed system with three replicas ($R1, R2$ and $R3$); $R1$ stores information regarding objects x, z, $R2$ objects x, y and $R3$ objects y, z. Each replica maintains causal information (e.g., a vector clock per object) about the objects it is interested in and only propagates updates to replicas that are interested in said object. For the sake of the example, let's also consider that all updates causally depend on all previous updates of the originating replica (i.e, clients read all objects before performing an update). $R1$ updates object x (update $up1$) and z (update $up2$) and then propagates the updates to replicas $R2$ and $R3$ respectively. Update $up2$ is causally dependent on update $up1$. If replica $R3$ updated object y (update $up3$) there is a possibility that the information about this update reaches replica $R2$ before update $up1$. Since replica $R2$ does not store any information about object z it would not be able to see the causal dependency that exists between update $up3$ and the in-flight update $up1$. This would result in replica $R2$ mistakenly applying it, when it should wait for the arrival of update $up1$.

As we can see, causal consistency under partial replication can raise a different set of challenges that do not happen with full replication.

2.3.1.B An Overview of Current Solutions

As seen in Section 2.3, several mechanisms have been developed to track causality. While efficient in representing the causality dependencies, these mechanisms typically either suffer from a lack of scalability due to the size of the metadata or lose information about causal dependencies, which can lead to false dependencies between concurrent updates. In partially replicated scenarios, keeping the size of the metadata lean is key for the performance of the system. When designing a causally consistent system, these issues need to be taken into account. We now take a look at some approaches that enforce causal consistency in partial replication setting.

PRACTI [38]: PRACTI is a replicated system that is able to simultaneously achieve partial replication (PR), arbitrary consistency (AC), and topology independence (TI). In this system, replicas can choose each individual object that they will replicate. These objects are then added to the *interest set* of the replica, which is free to change its contents at any given time. Apart from the interest set, each replica maintains a version vector with one entry per replica. Updates and reads are executed locally at each replica and propagated afterward. Each update is tagged with the current value of the node's Lamport clock and the node's ID. PRACTI propagates updates in a similar way to Bayou's [39] log exchange protocol; however, it introduces a novel mechanism; the idea is to separate the propagation of the updates' metadata from the actual values set up by the update. This separation of responsibilities gives PRACTI some freedom regarding the way metadata and data flow through the system. While update messages (named *Body messages*) can be delivered in any order to the replicas, the metadata propagation channels (named *Invalidation streams*) must respect a causal ordering of delivery. Invalidation streams deliver two types of messages: *precise invalidations* and *imprecise invalidations*. Precise invalidations correspond to the metadata regarding single updates. Imprecise invalidations represent multiple ordered precise invalidations and act as a summary of this group of messages. To guarantee correctness, all replicas must maintain all invalidation messages. By doing so, they can control the arrival order of the Body messages and apply them according to the current consistency policy of the system - e.g if the system is guaranteeing causal consistency then replicas need to wait for the arrival of the dependent Body messages before applying one.

Saturn [37]: Saturn is another system that decouples the metadata from the actual data propagation. It is a metadata service for geo-replicated systems; therefore, it focuses only on metadata management and assumes that data is propagated using an existing bulk-data mechanism that fits the application business requirements. Saturn works with small-size metadata that takes the form of *labels*. Labels work as tags for updates and are generated by each server. They can be ordered through their timestamps that are generated from a physical clock at each server. Inside each datacenter exists a centralized component called the *label sink*. This component is responsible for gathering all labels, ordering, and then propagating them to the inter-dc module of Saturn. Clients attach to a certain data center

in order to communicate with the system. If a client desires to change datacenters, it must request a migration to Saturn. While Saturn introduces protocols to ensure clients respect causality, the most interesting mechanism lies in how Saturn propagates labels between datacenters that lets it achieve genuine partial replication: *Labels are distributed according to a fixed dissemination tree between data-centers*. In this tree topology, data centers act as leaves, and the nodes leading up to those leaves are servers named *Serializers*. Serializers Inter-communication and communication with the datacenters are done through FIFO channels. This allied to the tree dissemination is sufficient to guarantee causal consistency. Another advantage with the use of a tree is that serializers do not need to propagate labels through branches that will lead to data centers that do not replicate the item associated with the label. This mechanism is fundamental in the sense that it enables genuine partial replication. The use of a tree can raise some challenges. In the case of a failure in a node of the tree, it needs to be recomputed, which is a time-consuming process.Another problem is the fact that every label must be propagated through the tree, which can become a bottleneck in the system. One system that tries to mitigate these issues is C^3 [40]. Like Saturn, C^3 also separates the metadata from the data propagation; however, the information is propagated directly between every pair of nodes. Not considering a tree raises the need for a larger amount of metadata to be tracked (timestamps increase from a single scalar to a vector with one entry per data-center), and genuine partial replication cannot be achieved.

Swiftcloud [41]: A tradeoff that appears to be predominant in causally consistent systems is the throughput at which writes can be applied versus data visibility latency [42]. Swiftcloud provides fast writes and reads at the expense of clients having to access more stale data. In this system, clients maintain a local cache to which they apply updates and later send them to possibly different datacenters. The effect of each update only becomes visible once the operation is replicated in K datacenters (K ¿= 1); in the case of a fault, this allows clients to migrate to other datacenters without being rejected (a rejection would happen in the case where clients read a more up-to-date version of the data and the new datacenter cannot communicate with the faulty datacenter). While clients partially replicate information, data centers are fully replicated. Each client maintains a set of objects in its interest set to which they can write immediately. Datacenters will constantly be pushing notification updates to the clients regarding those objects. Clients, on the other hand, will constantly be sending messages regarding unacknowledged updates to the datacenters. The communication channel is FIFO, and clients must maintain a session opened with the first node they contact. To guarantee convergence and solve conflicts, Swiftcloud's clients use a library of CRDT objects. When clients need to read an object, they do so in a causally coherent state that is exposed by the datacenter. This state may be stale on some occasions, but by exposing a coherent state, the system avoids the possibility of clients generating causal gaps due to the clients' access patterns. In terms of the metadata, Swiftcloud compresses causal dependencies into a vector clock with an entry per data-center.

PaRiS [43]: In PaRiS, a transactional causally consistent system that also supports partial replication, clients also maintain a local cache and read from a coherent snapshot computed through the use of a novel causality tracking mechanism named *Universal stable time* (UST). The snapshots are such that they have been installed in every datacenter; therefore, and unlike Swiftcloud, reads do not block. Other approaches also rely on clients reading from a stable snapshot [44, 45]. Regarding metadata, PaRiS uses Hybrid logical clocks [19] complemented with two vector clocks, for each partition, to keep track of current partition timestamp and other to keep track of the global stabilization time.

Legion [46]: Legion is a framework that allows client web applications to easily replicate data from the servers and between them in low latency and secure manner. Communication between clients can be done directly without the need to use a server as a middle-man. Since clients' devices are limited in storage, Legion adopts a client-side partial replication approach. In Legion, objects are stored in containers. Each container has an associated multicast group to which clients can join if interested in the objects in the container. Legion opts to use a vector clock with one entry per replica of a container. This means that more popular containers will have bigger metadata. Locally, at each client, objects are modified using a library of CRDT objects, and updates are transmitted between nodes through a FIFO channel, which allied to the multicast primitive guarantees causal delivery of updates. All reads and writes are applied to the versions of the objects that are locally available. When performing an update, the client creates a new object version that is timestamped with the current clock value.

Karma [47]: Karma partially replicates data at the datacenter level. Groups of geographically close datacenters form a consistent hashing ring and in which data is replicated. Karma ensures that causality is guaranteed inside each ring; however, between ring updates are propagated asynchronously. To avoid clients breaking causality, the system keeps track of in-flight messages and blocks the client from reading from a different ring until the system deems it to be safe. Regarding the type of metadata used, Karma tracks dependencies with great detail using a approach similar to the one used by COPS [25]. It tracks dependencies at the granularity of data items.

Discussion: Comparing the different systems, one can notice some characteristics of partially replicated causally consistent systems. Regarding where the partial replication happens, we can divide the systems into two types: client-level and server-level. Client-level systems replicate data on the client's devices; systems such as Swiftcloud, PaRiS and Legion maintain a cache on each client. Server-level systems partially replicate data between the different nodes of the system; The level of replication can go from simple replica servers (PRACTI) to datacenters (Saturn).

A predominant approach in these systems is to let clients read from data that that is potentially stale. Some systems, such as PaRiS and Saturn, expose a consistent snapshot to clients from which they can read. Another approach is the one proposed by SwiftCloud where a client observes a certain update only if it is stored in K different replicas (k-staleness).

Another characteristic of these systems is the topology of the communication of replicas. PRACTI and Legion set no restriction on this topology; however, by limiting the topology of the system, one can introduce some mechanisms (the tree topology in Saturn and the consistent hash rings in Karma) that are not possible in a more generalized scenario.

The separation of the propagation of metadata from the actual data is another interesting mechanism. This approach gives the system more flexibility on how metadata is handled as opposed to other systems that require metadata to be piggybacked inside each message. Systems that take advantage of this technique are PRACTI, Saturn, and C^3.

Regarding the type of metadata used by each system, as we have seen, most systems use some form of logical clock to keep track of causal dependencies. These logical clocks are mostly based on the ones that were presented in the previous Section. One of the major differences between systems is the semantic attributed to each logical clock. This semantic dictates how the clocks are used to enforce causal consistency and also the amount of metadata used. The fact that a lot of systems use some form of logical clock shows how important it is to be able to correctly decide on the metadata representation, which is the main focus of this thesis.

2.4 Causality tracking trade-offs

There is a considerable amount of work that addresses causal consistency, however, to the best of our knowledge, only a small part of this work provides an analysis of the trade-offs imposed by the different causality tracking mechanisms.

Bravo [11] makes a thorough analysis of the numerous systems that support causal consistency and shows that there is a direct correlation between the metadata size and the number of false dependencies, even for systems with optimized mechanisms. This correlation stems from the fact that when we compress the metadata, we fuse together sources of potential concurrency, which in turn increases the number of false dependencies. An intuitive example of this phenomenon is when we compress all the entries of a vector clock into a single Lamport clock. If we were to perform this compression, independent events (concurrent) happening at two different servers would now have a dependency between them which in reality would not exist. This false dependency would result in each server losing concurrency because the operation would not be able to be immediately applied without the arrival of its false dependency. This would increase the time it takes to apply a certain update.

The extra time it takes to apply an update after it has arrived at the destination server is named *remote visibility latency* and, as we have seen, is related to the level of concurrency offered by the type of metadata used. Having high visibility latencies has a significant impact in the system in two ways: first, users will observe increasingly staler data as visibility latency increases, which is not desirable for

systems that depend on data freshness; Secondly, clients performing migrations will have to wait longer, since their view of the data is not yet consistent with the server they are trying to connect.

Bravo also raises awareness to the fact that efficient causal consistency in partially replicated scenarios is difficult to achieve. This difficulty is, in fact, a trade-off between minimizing the amount of metadata being handled and the loss of concurrency by minimizing said metadata. On the one hand, having more metadata means that we can more precisely track causal dependencies. Although this may result in fewer false dependencies, it also costs more CPU, storage, and bandwidth, which in some systems may not be highly available. On the other hand, having less detail in the metadata may not be possible in partial replication scenarios as we have seen in Section 2.3.1.A.

Cheriton and Skeen [48] perform a very detailed exposition of the limitations of causally ordered communication and correlate visibility latency with the amount of buffering of updates due to missing dependencies. Still from a more theoretical point of view, Bailis et. al. [42] extend this analysis in the context of causal consistency. Their work also identifies throughput (i.e. the rate at which clients generate new updates) and visibility latency as competing goals. Additionally, they raise awareness to the poor scalability of the mechanisms used to ensure causal consistency.

While our work focuses on studying the trade-offs of logical clocks, Bravo et. al. [49] make an interesting analysis between the advantages of using different types of clocks to enforce causal consistency by analyzing different existing systems. Their observations address the impact of the client's access patterns on the performance of systems that use logical clocks, while also presenting physical clocks as an alternative to avoid some of the limitations of logical clocks. Although they may potentially increase performance on some scenarios, physical clocks still require that data stores inject delays in order to synchronize clocks across the system. Their work addresses the fact that time is, as well as other characteristics of a system, also a resource that can be traded for other resources. This means that the overall performance of a given algorithm is hard to predict, and strongly depends on which resources the system is more constrained on. For example, if the system is constrained in bandwidth, having a lean metadata representation may or may not result in better performance, it highly depends on how much bandwidth is actually available.

Interestingly, they make the observation that fully decentralized systems may be as little scalable as fully centralized ones. They also identify the issue that the use of different clock implementations may highly depend on factors such as the type of workload, the hardware that is used in the system, network properties, etc. This makes it hard to define when it is better to use each type of metadata representation. These types of trade-offs is what we explore in this thesis, by trying to identify which workload patterns favor a certain type of metadata.

The analysis, in all these studies, is based on simplified models that fail to capture the interactions among the different parameters that characterize the workload of causally consistent storage systems.

To the best of our knowledge, our work is the first to identify a set of features that help to understand the performance of different mechanisms. Our study is also the first to make an experimental assessment of how different metadata choices affect the remote visibility latency in distributed storage systems.

2.5 Summary

This chapter presented the concept of time applied to distributed systems. We exposed the difficulties of keeping track of time and different approaches to achieve said goal. We focused on logical time, which goes hand in hand with the concept of causal order, and studied how we can keep track of causal dependencies. We then addressed causal consistency, a useful data consistency model that stems from the happened-before relation. Finally, we studied some of the trade-offs that exist when providing causal consistency in a system and how different data structures may affect the performance of the system.

3

Metadata management and costs

Contents

As we have seen in the previous Chapter, several algorithms and systems have been proposed to achieve causal consistency under partial replication. These algorithms are highly optimized and are often specialized to specific system conditions. In this Chapter, we present a generic framework that allows instantiating several causal multicasting algorithms using different data structures to track causality. Additionally, we also present a generic causal storage algorithm that builds upon the causal multicasting framework. Finally, we discuss some instances of the framework that are relevant and that will be used as a baseline for our future discussions. We give a theoretical analysis of the metadata costs associated with each one of the algorithms.

This Chapter is organized as follows: Section 3.1 presents the generic framework to provide causal multicast; Section 3.2 provides the generic causal storage algorithm, and Section 3.3 describes the relevant algorithms and metadata costs.

3.1 Attaining Causal Multicast

This Section presents a generic algorithm to implement causal multicast. The algorithm, depicted in Alg. 1 can be adapted to use different types of clocks. The algorithm assumes that multiple multicast groups can exist in the system but that causality is maintained across groups, similarly to what was provided by the Isis system [50]. Groups can be mapped to application-level abstractions, such as distributed objects. Later in Section 3.2, we show how groups can be used in the context of distributed storage systems.

The algorithm assumes that, for each group in the system, there is a set of reliable First-In First-Out (FIFO) channels connecting every pair of processes. These FIFO channels are used exclusively to send messages for that group. FIFO channels can be trivially achieved in practice, for instance, by using Transmission Control Protocol (TCP)/IP connections to support the message exchange. Each process p_i maintains a logical clock denoted *local_clock*$_i$. The format of this clock is implementation-dependent: it can be one Lamport clock, one vector clock, one matrix clock, or even a set of clocks (one for each group). The initialization of this clock is encapsulated by the primitive INIT_CLOCK.

For sending a message m, the process starts by updating its local clock in function PREPARE. The message m carries a logical clock, denoted *clock*$_m$, with the value of the local clock. The message is sent, using the FIFO channels, to the set of destination peers, typically the set of all processes in the group. Additionally, the message is locally delivered.

When receiving a message m for group g from process j, the process adds the message to the list of pending messages for g and j, *pending*$_i[g][j]$. A message that reaches the top of the pending list for some process p_j and group g (i.e. for which prior messages from p_j to g have already been delivered) can be delivered if its dependencies are satisfied, i.e., if all dependent messages have already been

Algorithm 1 Generic Causal Order Implementation for process P_i

```
1: Let 𝒩 be a set of processes
2: Let 𝒢 be a set of multicast groups
3:
4: procedure INIT_NODE(i)
5:     INIT_CLOCK (local_clock_i);
6:     delivered_upto_i[g][j] ← 0; ∀g ∈ 𝒢, ∀j ∈ 𝒩
7:     pending_i[g][j] ← ∅; ∀g ∈ 𝒢, ∀j ∈ 𝒩
8:     my_groups_i ← groups to which process i belongs
9:
10: procedure SEND(m, g, destination_set)
11:     PREPARE(g, destination_set)
12:     clock_m ← local_clock_i;
13:     for k ∈ destination_set do
14:         FIFO_SEND (m, clock_m, g, i, k);
15:     DELIVER(m, clock_m, g, i);
16:
17: procedure FIFO_RECEIVE(m, clock_m, g, j)
18:     ENQUEUE(pending_i[g][j], ⟨m, clock_m⟩);
19:
20: when ∃m : FIRST(m, pending_i[g][j]) do
21:     MESSAGE_READY (m, clock_m, g, j);
22: done
23:
24: when ∃m : FIRST(m, pending_i[g][j]) ∧ MESSAGE_SAFE(m, clock_m, g, i, j) do
25:     UPDATE_LOCAL_CLOCK(m, clock_m, g, j);
26:     DELIVER(m, clock_m, g, j);
27: done
```

delivered – this is checked in function MESSAGE_SAFE. For helping in this check, each process p_i keeps a record, *delivered_upto*$_i[g][j]$, with the largest clock for which a message from p_j to g has already been delivered in p_i or for which it is known that such message does not exist. This information is updated both when a new message is delivered (function UPDATE_LOCAL_CLOCK), and when a new message reaches the top of the list of pending messages (function MESSAGE_READY). This latter case is used to register that messages with smaller clocks from p_j to g do not exist – this follows from using FIFO channels: if an undelivered message with a smaller clock existed, it would be the message on the top of pending messages.

3.1.1 Causal Order with Lamport Clocks

Alg. 2 shows the functions to instantiate the generic algorithm for the case where Lamport clocks are used to keep track of causality. In this case, for each group g, each process keeps a Lamport's clock that is initiated to 0 (function INIT_CLOCK).

When sending a message, the local Lamport's clock associated with the group is incremented (function PREPARE). Note that the message is timestamped with the full local clock, which includes the set of clocks for all groups to trace dependencies across groups.

The MESSAGE_READY function, called when a message m reaches the top of the pending list from process p_j for group g, updates $delivered\_upto$ record to register that all messages with smaller Lamport

Algorithm 2 Causal Order with Lamport Clocks for process P_i

```
1: procedure INIT_CLOCK(clock)
2:     clock[g] ← 0, ∀g ∈ G;
3:
4: procedure MERGE_TS(ts_1, ts_2)
5:     result_ts[g] ← MAX(ts_1[g], ts_2[g]), ∀g ∈ G
6:     return result_ts
7:
8: procedure PREPARE(g, destination_set)
9:     local_clock_i[g] ← local_clock_i[g] + 1;
10:
11: procedure MESSAGE_READY(m, clock_m, g, j)
12:     delivered_upto_i[g][j] ← clock_m[g] − 1
13:
14: function MESSAGE_SAFE(m, clock_m, g, i, j)
15:     condition_1 ← clock_m[g] − 1 ≤ delivered_upto_i[g][k], ∀k ∈ N : k ≠ j
16:     condition_2 ← clock_m[g'] ≤ delivered_upto_i[g'][k], ∀g' ∈ my_groups_i : g' ≠ g, ∀k ∈ N
17:     return condition_1 ∧ condition_2
18:
19: procedure UPDATE_LOCAL_CLOCK(m, clock_m, g, j)
20:     delivered_upto_i[g][j] ← clock_m[g]
21:     local_clock_i ← MERGE_TS(local_clock_i, clock_m)
```

clocks from p_j to g either have been delivered or do not exist (this follows from the use of FIFO channels, as explained before).

The MESSAGE_SAFE function verifies that a message for group g is safe to be delivered by checking that all of its dependencies have already been delivered. This can be assessed by checking that: i) all messages for g with smaller clocks from all other processes have already been delivered ($condition\_1$); ii) the dependencies for other groups the local process belongs to have already been delivered ($condition\_2$).

The UPDATE_LOCAL_CLOCK function, called when a message is delivered, updates $delivered\_upto_i[g]$ to reflect the delivered message, and the local clock by taking the maximum of the local clock and the message's clock for each group. This guarantees that, when sending a new message, the message clock for the group will be larger than the clocks of all dependencies, which makes the use of the conditions defined in MESSAGE_SAFE correct.

3.1.2 Causal Order with Vector Clocks for process P_i

A well known extension of Lamport clocks are vector clocks [6, 12]. As presented in Section 2.2.2 a vector clock keeps multiple logical clocks, one for each process in the system, precisely recording the last message from each process - e.g., if the system has N processes, the vector clock will contain N logical clocks. Alg. 3 presents the functions to instantiate the generic causal multicast algorithm for vector clocks.

The implementation of these functions is conceptually similar to the corresponding implementation for Lamport clocks, described in the previous section. There are two major differences. First, for each

Algorithm 3 Causal Order with Vector Clocks for process P_i

```
1: procedure INIT_CLOCK(clock)
2:     clock[g][k] ← 0, ∀g ∈ G, ∀k ∈ N;
3:
4: procedure MERGE_TS(ts_1, ts_2)
5:     result_ts[g][k] ← MAX(ts_1[g][k], ts_2[g][k]), ∀g ∈ G, ∀k ∈ N
6:     return result_ts
7:
8: procedure PREPARE(g, destination_set)
9:     local_clock_i[g][i] ← local_clock_i[g][i] + 1;
10: procedure MESSAGE_READY(m, clock_m, g, j)
11:     delivered_upto_i[g][j] ← clock_m[g][j] − 1.
12:
13: procedure MESSAGE_SAFE(m, clock_m, g, i, j)
14:     condition_1 ← clock_m[g][k] ≤ delivered_upto_i[g][k], ∀k ∈ N : k ≠ j
15:     condition_2 ← clock_m[g'][k] ≤ delivered_upto_i[g'][k], ∀g' ∈ my_groups : g' ≠ g, ∀k ∈ N
16:     return condition_1 ∧ condition_2
17:
18: procedure UPDATE_LOCAL_CLOCK(m, clock_m, g, j)
19:     delivered_upto_i[g][j] ← clock_m[g][j].
20:     local_clock_i ← MERGE_TS(local_clock_i, clock_m);
```

group, we keep a vector clock instead of a Lamport clock, with each entry in the vector clock initiated with 0; when a message is sent on a group g, only the ith entry in g's vector clock is incremented. Second, as a message records precisely its causal past, the function that assesses if it is safe to deliver a message can verify that, for each process, the dependencies have already been delivered.

3.1.3 Causal Order with Matrix Clocks

Matrix clocks [7] expand vector clocks in an extra dimension. Instead of maintaining one logical clock for each process in the system, matrix clocks maintain one logical clock for each link in the system, allowing a process to know not only the messages received from each of the other processes, as with vector clocks, but also what messages each of the other process has received, allowing to track precise information about indirect dependencies.

When processes are logically organized in a clique, and every process has a unidirectional link to every process in the system (including to itself), the number of links is quadratic. In Alg. 4 we present the functions to support causal ordering using matrix clocks. The structure of the code is, again, very similar to the two implementations presented before, for Lamport and vector clocks, with the exception that the clocks carry much more detailed information, in particular, the causal past of a node (or message) is captured by the last message observed on each link. Sending a message on group g, updates the ith vector clock by incrementing all kth entries of said vector, where k are all processes that will receive the update message.

Algorithm 4 Causal Order with Matrix Clocks for process P_i

```
1: procedure INIT_CLOCK(local_clock_i)
2:     local_clock_i[g][j][k] ← 0, ∀g ∈ G, ∀j, k ∈ N;
3:
4: procedure MERGE_TS(ts_1, ts_2)
5:     result_ts[g][j][k] ← MAX(ts_1[g][j][k], ts_2[g][j][k]), ∀g ∈ G, ∀j, k ∈ N
6:     return result_ts
7:
8: procedure PREPARE(g, destination_set)
9:     local_clock_i[g][i][j] ← local_clock_i[g][i][j] + 1, ∀j ∈ destination_set;
10:
11: procedure MESSAGE_READY(m, clock_m, g, j)
12:     delivered_upto_i[g][j] ← l(m)[g][j][i] − 1.
13:
14: procedure MESSAGE_SAFE(m, clock_m, g, i, j)
15:     condition_1 ← clock_m[g][k][i] ≤ delivered_upto_i[g][k], ∀k ∈ N : k ≠ j
16:     condition_2 ← clock_m[g'][k][i] ≤ delivered_upto_i[g'][k], ∀g' ∈ my_groups : g' ≠ g, ∀k ∈ N
17:     return condition_1 ∧ condition_2
18:
19: procedure UPDATE_LOCAL_CLOCK(m, clock_m, g, j)
20:     delivered_upto_i[g][j] ← clock_m[g][j][i].
21:     local_clock_i ← MERGE_TS(local_clock_i, clock_m);
```

3.2 Causal Storage

As we have done for causal multicast, it is possible to derive a generic algorithm to ensure causal consistency in a distributed storage system. The algorithm is presented in Alg. 5 and it builds on the generic causal multicast algorithm introduced in the previous Section.

The extensions of Alg. 5 with regard to Alg. 1 are mostly focused on tracking and handling each client's read and write dependencies (variable *client_clock*). When the client performs a write, a new timestamp is associated with the update. This timestamp is computed by combining the value of the *client_clock* with the clock of the storage node to which the client is attached. Updates are afterward multicasted to all nodes that replicate that object. These updates are delivered in causal order and are kept pending at remote nodes until its safe to apply them. When an update is delivered, the value written by the client is merged with the locally stored object value and the corresponding timestamps are also merged. Note that, similarly to Alg. 1, Alg. 5 can also be instantiated to use Lamport, vector, or matrix clocks: this can be achieved by selecting the appropriate implementations of the INIT_CLOCK, MERGE_TS, PREPARE, MESSAGE_READY, MESSAGE_SAFE, and UPDATE_LOCAL_CLOCK procedures from Alg. 2, Alg. 3, or Alg. 4, respectively.

The algorithm also assumes the existence of a MAP_KEY_TO_GROUP function that can map objects to the groups used by the multicast algorithm. The next Section discusses how object keys can be mapped to groups.

Algorithm 5 Generic Causally Consistent Storage Implementation for process P_i

```
1: Let 𝒩 be a set of processes
2: Let 𝒢 be a set of multicast groups
3: Let 𝒦 be a set of object keys
4:
5: procedure INIT_NODE(i)
6:     INIT_CLOCK (local_clock_i);
7:     delivered_upto_i[g][j] ← 0; ∀g ∈ 𝒢, ∀j ∈ 𝒩
8:     pending_i[g][j] ← ∅; ∀g ∈ 𝒢, ∀j ∈ 𝒩
9:     my_groups_i ← groups to which process i belongs
10:
11: procedure INIT_CLIENT(client_clock)
12:     INIT_CLOCK(client_clock)
13: procedure READ(client_clock, k)
14:     ⟨value, ts⟩ ← STORAGE_READ(k)
15:     client_clock ← MERGE_TS(client_clock, ts)
16:     return(value)
17:
18: procedure WRITE(client_clock, k, value)
19:     g ← MAP_KEY_TO_GROUP(k)
20:     PREPARE(g, REPLICAS(k))
21:     update_clock ← MERGE_TS(client_clock, local_clock)
22:     for j ∈ replicas(k) do
23:         FIFO_SEND(⟨k, value⟩, update_clock), g, i, j)
24:
25: procedure FIFO_RECEIVE(m, clock_m, g, j)
26:     ENQUEUE(pending_i[g][j], ⟨m, clock_m⟩);
27:
28: when ∃m : FIRST(m, pending_i[g][j]) do
29:     MESSAGE_READY (m, clock_m, g, j);
30: done
31:
32: when ∃m : FIRST(m, pending_i[g][j]) ∧ MESSAGE_SAFE(m, clock_m, g, i, j) do
33:     DELIVER(m, clock_m, g, j);
34:     UPDATE_LOCAL_CLOCK(m, clock_m, g, j);
35: done
36:
37: procedure DELIVER(⟨k, new_value⟩, new_ts, g, j)
38:     ⟨old_value, old_ts⟩ ← STORAGE_READ(k)
39:     merged_value ← MERGE_UPDATE(old_value, new_value)
40:     merged_ts ← MERGE_TS(old_ts, new_ts)
41:     STORAGE_WRITE(k, ⟨merged_value, merged_ts⟩)
```

3.3 Metadata Costs

The implementation of causally consistent storage proposed in the previous Section, captured by Alg. 5, can be configured to use different amounts of metadata. There are two main mechanisms that affect the amount of metadata required by the algorithm. The first is the type of clocks that are used to keep track of causality, i.e., if the algorithm is instantiated to use Lamport clocks, vector clocks, or matrix clocks. The letters “L”, “V”, and “M” are used to identify each of these alternatives. The other is how to map object keys to multicast groups. In this thesis, only two scenarios are considered, namely: updates for all objects are propagated using a single multicast group (we identify this option by the prefix “1”) or each object uses a different group, of its own, to propagate updates (this option is identified by the prefix “k”).

The two mechanisms can be combined in different ways, as depicted in Table 3.1. For example, it is

Table 3.1: Metadata Configurations

		One single group in the system 1		One group per object K	
Lamport clock	L	1L	$O(1)$	kL	$O(K)$
Vector clock	V	1V	$O(N)$	kV	$O(KN)$
Matrix clock	M	1M	$O(N^2)$	-	-
N < K		1L < 1V < 1M < kL < kV			

possible to use Lamport clocks to track causality, and keep a different clock for each object (configuration named "kL") or to use a vector clock to track of causality but maintain a single vector clock for the entire system (configuration "1V"). These mechanisms address orthogonal aspects of the system's operation. The choice between Lamport, vector, or matrix clocks allows to capture different levels of detail about the *processes* that produce updates and the *processes* that are going to receive the update. The choice of the mapping function allows to capture different amounts of detail about which objects have been targeted by each update. To illustrate this fact consider the following examples:

- Consider a system with 4 process, p_1, ..., p_4, using kL and a process p_3 that receives an update u from p_2 with timestamp 5 for object o without having received previously an update for o with timestamp 4 from p_1. On one hand, because each object uses its own clock, the process p_3 knows that the missing update is for object o and not for some other object. On the other hand, because Lamport clocks are being used, p_3 cannot know which other process(es) did generate the update(s) in the past of update u. Due to this lack of detail, p_3 needs to wait for an update from both p_1 and p_4 before delivering u.

- Consider a system that uses 1V and a process p_3 that receives an update u from p_2 with timestamp $[4, 1, 0, 0]$ for object o without having received previously an update from p_1 with timestamp $[4, 0, 0, 0]$. On one hand, because vector clocks are used, it knows that the missing update must be received from p_1, and not from p_4. However, because a single vector is used for all objects, p_3 cannot guess which object generated the missing update. Due to this lack of detail, p_3 may be forced to wait for an update from p_1, even if update with timestamp $[4, 0, 0, 0]$ was performed on an object not replicated by p_3.

To avoid both types of false dependencies, illustrated by the examples above, one may use one vector clock for each different object (configuration kV). Unfortunately, as Table 3.1 shows, this is the most expensive configuration (note that, in most practical systems, the number of objects is much larger than the number of nodes in the system). Conversely, the configuration 1L is the configuration that uses less metadata, as a single Lamport clock is used to track causality for all objects. The table also shows how the other configurations compare to each other regarding the amount of metadata they require.

Note that, because updates are always propagated to all replicas of a given object, the configuration MO does not bring any advantages over kV, and therefore we do not consider it in our study.

3.4 Summary

This chapter presented a generic framework to achieve causal multicast which is independent of the data structures used to keep track of causal dependencies. We then presented three algorithms that build upon the mentioned framework to provide causal multicast using the classical data structures, namely Lamport clocks, Vector clocks, and Matrix clocks. Next, we proposed a generic algorithm for causal storage that, again, uses the causal multicast framework. Having presented the algorithms, we make a brief theoretical analysis of the possible costs of each instantiation and present the five algorithms that will be used in our empirical analysis. Finally, we described how the prototypes for the chosen algorithms were implemented.

4

Implementation

Contents

In this chapter we present the implementation details of the different chosen algorithms that will be considered in our evaluation. In order to fairly evaluate all algorithms, a common framework was built which allows the same conditions and code to be applied for each algorithm. Section 4.1 describes the technologies used in order to build the framework and run the simulations. Section 4.2 presents the implementation details of the framework. Finally, Section 4.3 describes how each individual algorithm was implemented using the described framework.

4.1 Development Environment

All algorithms are simulated using the PeerSim [51] simulator which allows the creation of specific system conditions, some of which are not possible to achieve in real-life systems. These conditions do not change between simulation sessions due to a seed system. This allows us to precisely measure the performance of each algorithm as the same interactions with the system are performed each time for a given seed. This is especially important in our experiences since different clients' interactions influence the outcome of each algorithm's performance. The prototype was implemented using the Java programming language (OpenJDK 14 [52]). Additionally, to ease the testing process, a setting system was built where specific scenarios can be written, loaded, and reproduced when testing different algorithms. This system allows to configure not only the characteristics of the system, as well as the placement of objects and the client's interactions with the system.

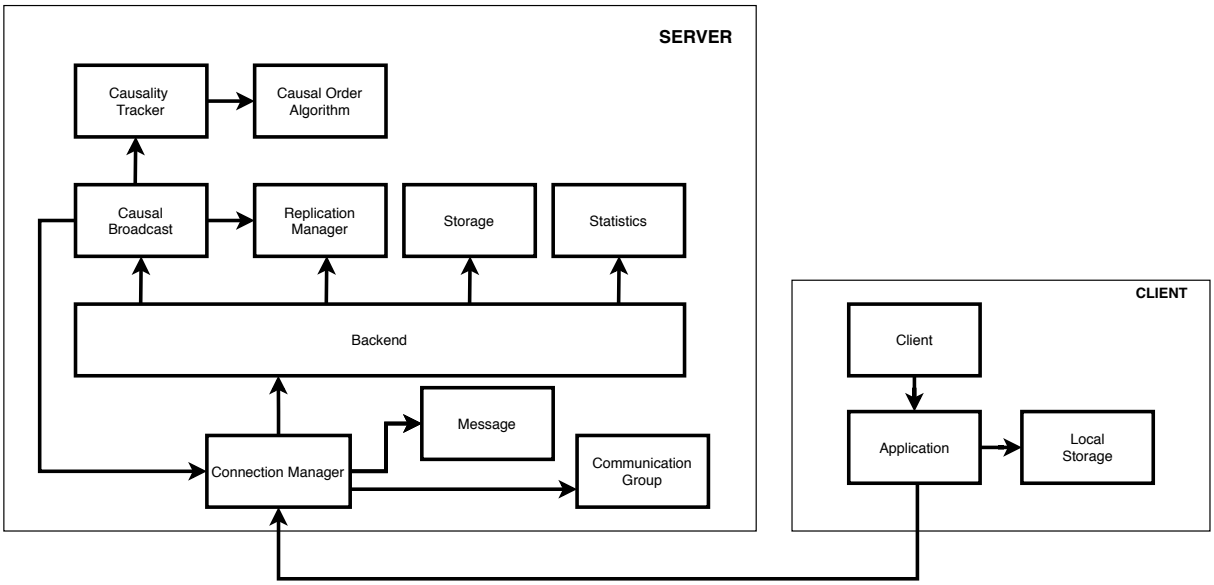


Figure 4.1: Framework components and interactions between components

4.2 Framework

We now describe in more detail the framework that was built in order to run the different algorithms. This framework tries to generalize as much as possible all of the code that is common and it allows all of the algorithms presented in Sections 3.1 and 3.2 to be instantiated. All the components and interactions of the framework are present in Figure 4.1. Overall, we divide the framework in two bigger components: the server and the client. The server is what is present at each node of the system; the client is the specific application that interacts with a single node. Now, we describe the components that do not need to be implemented when a new algorithm needs to be created.

4.2.1 Server Components

- **Storage:** The Storage component is a generic key-value storage used to store all the objects replicated locally at the node. The storage is accessed by the Backend to apply the updates that were executed or to read the current value of a certain object.
- **Replication Manager:** The Replication Manager is responsible for keeping track of which objects are locally replicated and which other nodes replicate a given object. The Backend and the Causal Broadcast component access the Replication Manager. The Backend registers where new objects are replicated, and the Causal Broadcast component consults the placement of objects in the system.
- **Causal Order Algorithm:** The Causal Order Algorithm is responsible for providing the data structures and the concrete algorithm implementation used to order the arriving messages. Every new algorithm that is added to the framework needs to implement this component. The specific algorithm that is to be used in the simulation, is decided through the use of a scenario configuration file.
- **Causality Tracker:** The Causality Tracker keeps track of the messages that are currently buffered (i.e. are not safe to be applied yet) and decides when a certain update message can be delivered to the Backend to be executed. It accesses the Causal Order Algorithm to causally order pending messages.
- **Causal Broadcast:** The Causal Broadcast component is responsible for providing causal multicast primitives that are used by the Backend in order to propagate updates. The Causal Broadcast accesses the Causality Tracker to order and store arriving messages before they can be applied. It is also used by the Backend to help decide when a certain message is safe to apply. Additionally, the Causal Broadcast accesses the Connection Manager in order to not only send messages using a FIFO connection, but to also get information regarding the different communication groups.

- **Statistics:** The statistics component has the responsibility of keeping track of all the relevant interactions that occur in the system and of extracting relevant statistics from them. As an example, the statistics component keeps track of the time it takes for a pending update to be applied in order to compute relevant metrics such as the Consistency Maintenance Overhead (CMO) (which will be presented in Chapter 5).

- **Connection Manager:** The Communication Manager is responsible for keeping track of all the existing connections to the server, for both clients and other servers. It is also responsible for providing FIFO channels for all the communications. The Communication Manager accesses the Message component, which varies depending on the Causal order algorithm being currently used. This component also accesses the Communication Group component to consult and register information about existing communication groups.

- **Message:** The Message component holds information that is sent between servers. This component is implementation-specific and depends on the implementation of the Causal Order Algorithm, as each algorithm needs to send different data structures and data in each message. As such, when creating a new Causal Order Algorithm, it is also necessary to build a specific Message component.

- **Communication Group:** The Communication Group tracks information about the existing communication groups.

- **Backend:** Finally, the Backend component is the core of the Server and is responsible for bridging the interactions between the other different components.

4.2.2 Client components

Regarding the client components, we have:

- **Client:** The client component provides an abstraction of a real person interacting with the system. Therefore, it can be configured to behave in different ways and to perform specific access patterns at different rates. This module interacts with the Application module to simulate a user interacting with an application.

- **Application:** The application provides a simple interface to interact with the Backend of a given server. When a Client updates an object, it propagates the update to the system and, upon return, applies the changes to the local storage.

- **Local Storage:** The Local Storage component stores the objects that are accessed by each client. Therefore, it serves the purpose of simulating the client's local device storage.

4.2.3 Conflict Resolution

Updates to objects are propagated asynchronously to other nodes. When we have two updates that are concurrent (i.e., we cannot establish which update happened first), the order in which they are applied may vary between nodes. When we consider updates to data, applying different updates in different orders may result in distinct states across the system. While under causal consistency this is valid, in this prototype both servers and clients use commutative or convergent replicated data types (CRDT [53]), a class of distributed data structures that can be replicated throughout different nodes and have two important properties:

(i) Updates to a certain replica can be done without any coordination and conflicts are automatically resolved

(ii) Replicas are guaranteed to eventually and deterministically reach the same state given that they saw the same set of updates

Using CRDTs allowed nodes to work as asynchronously as possible and to focus on the effects of different metadata structures without worrying about the performance impact of running conflict resolution algorithms.

4.3 Algorithms

This section details how the One Lamport clock per system (1L),one Lamport clock per object (kL), One Vector clock per system (1V), kL, and One Matrix clock per system (1M) algorithms were implemented using the framework described in the previous section. Because there are algorithms that share the same data structures, the implementations are similar, therefore, they will be described together.

4.3.1 1L and kL

The 1L and kL algorithms use as their main data structure a Lamport clock to keep track of causal dependencies. As described earlier, to add a new algorithm to the framework, we needed to implement a Causal Order Algorithm and Message components. Regarding 1L, each Causal Order Algorithm component keeps a Lamport clock structure which was implemented as a separate Java class. Additionally, it also keeps track of all highest seen Lamport clocks from other nodes which are represented as a java List of Lamport clocks. Regarding the message implementation, Lamport clocks are serialized as Java Integers. Since the 1L maps all communication groups towards a single entry (representing the entire system), the message does not need to carry additional information regarding which group the update happened.

Since kL also uses Lamport clocks, the 1L component was adapted to support multiple groups by keeping a map of Lamport clocks, one for each communication group. The same was done to the List of Lamport clocks. Regarding the message implementation, each message now holds a map of Lamport clocks, and an additional parameter that identifies the communication group was added. The communication group's identifiers are, in this algorithm, the keys of the objects.

4.3.2 1V and kV

Similarly, the 1V and one Vector clock per object (kV) algorithms use Vector clocks. Vector clocks are also implemented as separate Java classes that contain lists of Lamport clocks. Similarly to the 1L algorithm, the 1V algorithm keeps a single Vector clock to track the current highest known timestamp. Each server also keeps a vector clock for each object's version, together with the current clock of each client connected to the system. Regarding the message implementation, Vector clocks are serialized as lists of integers.

Again, the kV algorithm expands the 1V's Causal Order Algorithm component and adds support for multiple communication groups by keeping a map of all the previously mentioned structures, one for each object. Messages contain a map of lists of integers, as well as the identifier of the communication group (in this case object), which was the target of the update.

4.3.3 1M

Finally, the 1M algorithm keeps track of causal dependencies using a Matrix clock. The Matrix clock was also implemented as a separate Java class and is internally represented as a list of Vector clocks. Similarly to the 1V algorithm, each server keeps a single Matrix clock to keep track of the highest seen timestamp. It also keeps a Matrix clock for each object to track their versions, and a Matrix clock for each connected client. Matrix clocks are serialized as lists of lists of integers and do not need to send additional information regarding the communication group, since it is assumed to only exist one communication group.

4.4 Summary

This chapter presented a high-level overview of how the prototypes of the algorithms were implemented. First, the development environment was explained; Next, the components that make part of the framework used to build the prototypes were described; Finally, a more detailed explanation of how the algorithms were implemented was presented.

5

Empirical Study

Contents

While comparing the cost of the different metadata configuration is straightforward, to assess the benefits of using a more expensive solution is harder. This chapter presents a study that compares the performance of the different configurations under different workloads. Our goal is to characterize the scenarios where using more expensive metadata brings advantages that justify the additional bandwidth and storage space.

The chapter is organized as follows: Section 5.1 presents the goals of this empirical study; Section 5.2 describes the system settings in which simulations were run; Section 5.3 provides a description of the features and metrics considered in the study; Section 5.4 describes the tested scenarios; Section 5.5 presents a preliminary assessment of the possible costs and benefits that we can expect; Finally, in Section 5.6 the results of the experiment are presented.

5.1 Goals

The goal of this empirical study is to determine which algorithms are more suitable to be used depending on the workload of the system. We try to answer the following questions:

- Which features or combination of features of the system's workload favor each algorithm?
- In which scenarios can less metadata be used without affecting the system's performance?

To answer these questions, we compare all systems against each other considering various workloads. We also try to clearly characterize the features of each tested workload.

5.2 Experimental Setting

All experiments were performed using the PeerSim simulator [51], extended with a networking module that provides point-to-point First In, First Out (FIFO) channels, and allows network latency to be configured.

We consider a setting of $N = |\mathcal{N}|$ nodes that collectively store $K = |\mathcal{K}|$ objects. Storage nodes represent data centers placed in different geographical locations. The latency among nodes is configurable. Every object $k \in \mathcal{K}$ is partially replicated and stored in a subset of all nodes *replicas*$(k) \subseteq \mathcal{N}$, i.e., we considered a setting with partial replication. Full replication is a particular case of this general setup, where every node replicates all objects.

Each node has a fixed number of clients that perform read and write operations on objects replicated on that node. Clients execute a loop where they perform a sequence of 0, 1, or more reads followed by a single write. The number of reads that precede each write defines the read/write ratio of the workload. Clients select objects to read or write randomly according to some distribution. There is a

configurable think time between any two consecutive client operations, for simulating processing time, user interaction, and client-server latency.

The set of parameters that control our experiments, listed in Table 5.1, is the following:

- The total number of nodes is denoted N; in all experiments we have $N = 16$. Each node (server) has 10 clients.
- The network latency, among any pair of nodes (i, j), follows a Gaussian distribution with average $\overline{\delta}_{i,j}$. The averages are taken from the real latencies observed in AWS datacenters [54], with 6 nodes in Europe, 4 in North America, and 6 in Asia.
- The number of objects K is constant in all experiments and set to 1600.
- All objects have the same number of replicas, and we vary the number of replicas from 2 to N, with the latter corresponding to full replication.
- The number of objects stored at each node is uniform (i.e., all servers store $R \cdot K/N$ objects).
- Each server is configured to have 10 clients; Additionally, it is considered that clients join the system and start performing updates at different moments in time. This is dictated by the J_i variable that states the rate at which new clients join the system.
- Two scenarios were considered for the think time T_i used by clients. In the first scenario, the think time of all clients is constant, with all clients producing updates at the same pace. In the second scenario, clients of different nodes have different think times, according to the average (λ) of an exponential. This means that each individual client will produce updates at different rates.
- All clients use the same write-read ratio, that we set to 0.1.
- Finally, each object has a pre-determined probability $P_i(k)$ of being accessed by any client. This probability can follow either a uniform distribution where all objects in a certain server i have the same probability of being accessed, or the access pattern follows a Zipfian distribution where some objects are accessed more frequently than others.

5.3 Characterizing the Workloads

The large number of features that define a scenario, make the analysis of distributed storage systems particularly hard. In fact, it is infeasible to experiment with all possible combinations of the features enumerated in Table 5.1. Still, in this experimental work, it was possible to observe that some mechanisms exhibited the same performance in a multitude of apparently distinct scenarios. This observation is what

Table 5.1: Parameterization

Configuration	Variable	Distribution	Value
Number of nodes	N	-	16
Network latency	$\delta_{i,j}$	gaussian	$[30ms, 400ms]$ taken from [54]
Number of objects	K	-	1600
Replication degree	$R(k)$	constant	$[2, N]$
Objects per node	$S(n_i)$	uniform	$R \cdot K/N = R \cdot 100$
Clients per node	C_i	constant	10
Client join rate	J_i	gaussian	50ms
Client think time	T_i	constant exponential	$[10ms, 1000ms]$ $\lambda \in [10ms, 100ms]$
Read/Write ratio	rwratio	constant	$\frac{writes}{reads} = 0.1$
Object access pattern	$P_i(K)$	uniform zipfian	$1/S(n_i)$ $\alpha = 0.9$ (from [55])

motivated the search for a new set of features that can capture the properties of the scenarios that are relevant for the performance of the different causality tracking mechanisms.

Therefore, next, two novel features that help characterize the workload are proposed. As mentioned, these features have the purpose of understanding the impact of different metadata techniques. Recall that vector clocks capture information about which nodes have sent updates in the causal past and that the mapping function allows us to capture information about which objects were updated in the causal past.

5.3.1 Update Generation Rate Asymmetry (*Update Generation Rate Asymmetry (GRA)*)

Definition: Update generation rate asymmetry is defined as the ratio between the fastest and slowest average update frequency for all nodes in the system.

5.3.2 Intuition

The update generation tries to capture the level of asymmetry in the generation of updates in the system. The higher the GRA, the more asymmetric are the events happening at a given server (for example, one server has lots of clients accessing and another server has few). The smaller the GRA, the more uniform the system is (for example, all clients access the same object at different replicas at the same time).

5.3.2.A Computing GRA

Several parameters influence the computation of the GRA. In real-life systems, these parameters may be dynamic and, therefore, the GRA may vary accordingly. Nevertheless, in the following experiments, these parameters are fixed which allows an easier computation of this feature. Since the GRA is a ratio between the average update frequencies, we need to be able to compute the update frequency for a given node. Predicting when updates happen is not an easy task but can be approximated. In this thesis, the update frequency uf_i of a given node i is given by Formula 5.1.

$$uf_i = \frac{rwratio * C_i}{T_i} \tag{5.1}$$

Consider UF as the set of update frequencies for all nodes in the system. More precisely: $UF = \{uf_i | \forall i \in N\}$. Given set UF, we can compute the GRA of the system as follows:

$$GRA = 1 - \frac{min(UF)}{max(UF)} \tag{5.2}$$

The GRA formula will output values between 0 and 1. When the GRA is closer to 0 it means that both the $min(UF)$ and $max(UF)$ are similar and, therefore, the update generation rate asymmetry is low. In contrast, the further apart the min(UF) and max(UF) are, the closer to 1 will the *GRA* be and the update generation rate is more asymmetric.

5.3.3 Object Ownership to Objects in Causal Past Ratio (*Object ownership to objects in causal Past Ratio (OPR)*)

Definition: Given the set of updates received in a node, U, the causal frontier contains all updates $u \in U$ for which there is no other updates that happened after u, i.e., $\nexists u' \in U : u \rightsquigarrow u'$. The causal frontier defines the direct dependencies of an update, i.e., the updates that are relevant to track causality. We define OPR as the ratio between the objects with updates in the causal frontier and the number of objects replicated at a node.

5.3.4 Intuition

The OPR feature captures the composition of objects and replicas in the system as well as the access patterns to said objects. The OPR feature allows, for example, to say that two different systems will perform similarly given that the OPR is equal for the two systems. A small OPR value tells us that the system is, most likely, partially replicated, with few objects in common between replicas that are being accessed by clients. On the other hand, an higher OPR value means that the system is most likely fully replicated and clients access similar objects at different replicas.

5.3.4.A Computing OPR

The Object Ownership to Objects in Causal Past Ratio is a more complex feature to compute. An accurate way to compute this metric would be to, first, let the system run for a while and gather all messages passed between nodes. Next, for each message, compute the OPR based on the causal frontier of the timestamp and the objects replicated at the node that received the message. This approach, while accurate, has a few drawbacks. First, one needs to run the system for a while before being able to compute the feature. This means that while testing different system configurations one could not take into consideration the OPR without running said configuration first. Secondly, gathering all messages may be a time consuming, and storage and bandwidth costly process. Another way to compute the OPR would be to approximate it.

Instead of computing the OPR for each individual message, it is simplified by considering each network link. First, we approximate the number of objects that would be in the causal frontier of a message between two nodes. Given two nodes i and j, and the set of objects replicated at both nodes (respectively O_i and O_j), the approximated number of objects in the causal frontier for link $i \rightarrow j$ (ACF$_{ij}$) is given by the formula:

$$ACF_{ij} = sum(\{min(P_i(k) * C_i, 1) \mid \forall k \in O_i \cap O_j\}) \tag{5.3}$$

Function sum receives a set of numbers and adds up each element of the set. Function min chooses the minimum value between two numbers.

Finally, the average OPR for the system is given by:

$$OPR = avg(\{\frac{ACF_{ij}}{K_j} \mid \forall i \forall j\}) \tag{5.4}$$

Similarly to the GRA, this formula will also output values between 0 and 1. An OPR value closer to 0 means that for any server i that receives a message from server j, the causal past of said message will contain fewer updates to objects that are replicated by both i and j. If instead, the OPR value is closer to 1, then messages will contain more updates to objects in the causal past that both replicate.

5.4 Scenarios

Given the high number of variables in the system, it is neither possible to test all combinations nor is it relevant to consider all scenarios. To this end, only a few relevant cases were chosen. The scenarios that were considered in the empirical study are the following:

- **Variable *OPR* scenario**: In this scenario we vary the OPR and change the asymmetry of the system by considering multiple fixed GRA values.

- **Variable *GRA* scenario**: In this scenario we vary the GRA and increase the asymmetry of the system by considering different OPR values as well as the T_i and J_i distribution.
- **Highly uniform scenario**: In this scenario we consider an highly uniform system and vary the underlying network latency.

5.5 Costs and Benefits

This section discusses the costs and benefits of the different choices that are considered in this analysis.

The size of the metadata used by the different techniques was presented in Table 3.1. As noted before, metadata consumes network bandwidth and storage, because the metadata needs to be sent with each update and needs to be stored to be later retrieved. The larger the metadata, the more processing time and memory is needed to compare and merge clocks. In our experiments, we abstract from storage and CPU/memory costs and only take into account the bandwidth costs.

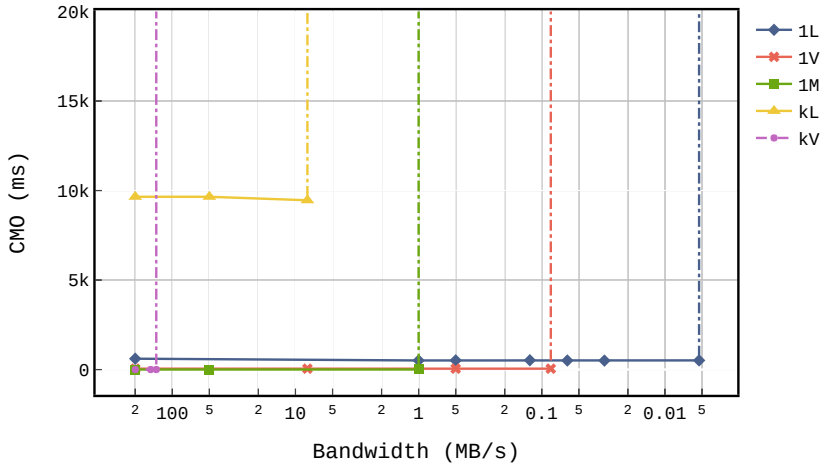


Figure 5.1: Bandwidth saturation: *CMO* 99th percentile; varying Bandwidth. $N = 16$; $K = 1600$; uniform think time $T(i) = 15$; uniform $R(k) = 5$; uniform access pattern; *GRA* $= 0.7$ *OPR* $= 0.2$

Naturally, if the deployment is constrained in terms of storage, processing, or bandwidth, the forms of causality tracking that consume more metadata may saturate the system. This is illustrated in Figure 5.1 that shows the point where different configurations saturate the links among the nodes. However, a more interesting problem is to understand if the techniques that uses more metadata brings benefits in an unconstrained system. Therefore, even if the system is not saturated, there is no reason to waste additional resources unless some benefits can be extracted. As such, in the remaining of the experi-

ments, it is always considered that the available bandwidth is enough to avoid link congestion when the target causality track is used. This allows us to assess if a mechanism can bring benefits when enough resources are available.

In this thesis we consider the *consistency maintenance overhead* (*CMO*) on remote update visibility latency as the benefit function. *CMO* is measured as the time it takes for an update u to be applied at node i *after* u has been received by node i. The smaller the value of *CMO* the better. As described, an update u received at a given node i can only be applied after the node knows that all updates in its causal past have been applied. As discussed, in this process, u may need to wait for a real dependency that is missing (because the system is asynchronous, messages can be delayed in the network) or may be delayed due to a false dependency, that results from the lack of detail of the metadata scheme. As schemes that use larger metadata are likely to exhibit less false dependencies, we expect schemes that use larger metadata to introduce fewer delays in applying remote updates, contributing to a smaller latency in the visibility of remote updates.

5.6 Experimental Analysis

This experimental analysis starts by observing how varying the *OPR* affects the *CMO* metric. All these experiments are presented in Figure 5.2. In this scenario, we start with a relatively uniform system and increase its asymmetry by considering different values of GRA; additionally, for each GRA value, we vary the OPR. By comparing Figures 5.2(a), 5.2(b), 5.2(c) and 5.2(d) we can mainly extract two interesting results: First, as we consider increasing GRA values, we can notice that the performance of 1V and 1L starts deteriorating. Second, as the OPR increases, independently of the GRA value, the CMO of both mentioned algorithms also decreases; In fact, the point from which this phenomenon appears to happen is when $\mathit{OPR} > 0.35$. This happens because as objects get more replicated the average *OPR* also increases, which in turn results in nodes receiving more updates with objects in their causal past that they also replicate. Regarding algorithms with more detail, the kL algorithm presents an unexpectedly poor performance as we can see in Figure 5.2(e), in fact as the OPR increases, the CMO also increases. We postpone the intuition on why the kL algorithm performs poorly to Section 6 and do not include kL in the rest of our experiments due to its general poor performance. As seen in Figure 5.2(f) both 1M and kV improve the performance as the OPR increases. Notice, that kV and 1M have similar behaviors when we change the OPR, but kV has slightly better performance; this is due to the fact that kV can use one FIFO connection per object, increasing the concurrency of each individual server. Keep in mind that this is only true considering we are not reflecting the effect of bandwidth in our experiments.

We now address the scenarios where we vary the GRA (Figure 5.3). As one may have already noticed, increasing the asymmetry of the system induces higher CMO values for 1V and 1L. We can

increase the asymmetry in various ways: different object access patterns; object placement; the number of replicas, etc. In Figures 5.3(a) and 5.3(b) we can observe that 1V performs especially well in full replication (*OPR*$= 1$), and despite the GRA varying, its performance is similar to the 1M and kV algorithm. Additionally, notice how the client join rate J_i, slightly affects the CMO of the 1L algorithm. As we further increase the asymmetry and consider a smaller value of OPR (Figures 5.3(c) and 5.3(d)) the 1V algorithm starts performing fairly bad. In fact, for GRA values larger than 0.7 the algorithm shows a shift in performance. An interesting phenomenon can be observed in Figures 5.3(e) and 5.3(f); the 1M and 1V algorithms seem to perform better as the GRA increases. This happens because updates are more diverse, meaning that most likely a certain FIFO connection will be less saturated with constant updates, resulting in a lower load for each link, and, consequently inducing lower CMO values. Additionally notice how for higher OPR values, the 1M and kL algorithms output lower CMO values overall.

It is clear that, if clients access the system at the same moment and have the same uniform think time, updates will be generated at the exact same rate. Intuitively, in such a scenario, one would expect the 1L algorithm to output *CMO* values closer to the 1V algorithm. However, this is not the case and the 1L algorithm still experiences a high *CMO*. This is an artifact of the asymmetry introduced by the latencies of the underlying network. In Figure 5.4 we can observe such relationship. Each server can only apply messages as fast as the slowest node generates them; since, in this case, every server generates new updates at the same rate, the asymmetry happens when messages take different times to arrive at the various servers. The 1L algorithm would only perform well if every server generates messages at the exact same rate and all messages arrive at every server at the same time, which is a highly unlikely scenario in real systems.

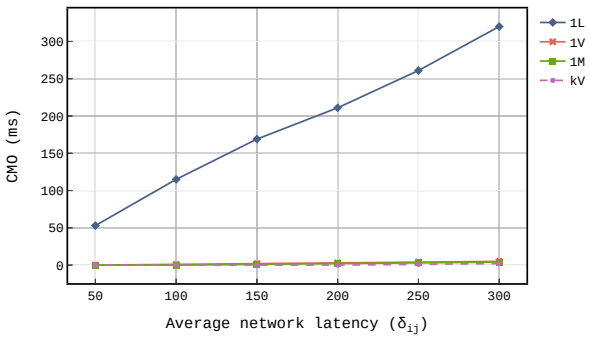
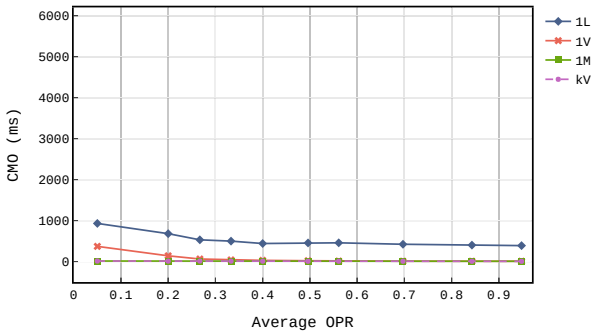


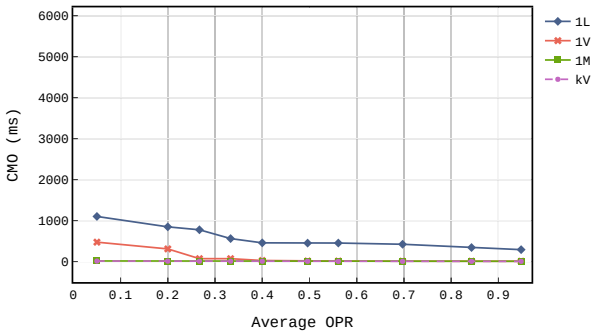
Figure 5.4: All uniform scenario: *CMO* $99th$ percentile; varying average δ_{ij}; $N = 16$; $K = 1600$; $J_i = 0$; uniform $T_i = 15$; $R(k) = N$; *OPR* $= 0$; *GRA* $= 0$; uniform $P(k)$. In this scenario, the worst case δ_{ij} is equal to the average δ_{ij}

5.7 Summary

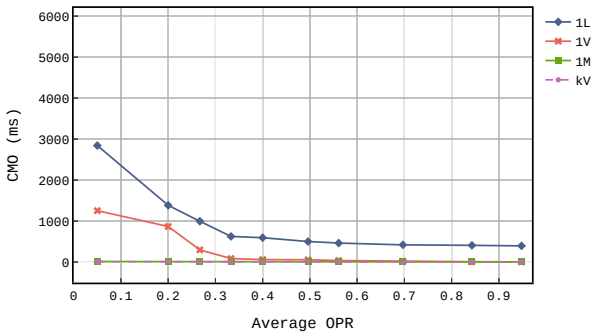
In this chapter, we provided a thorough evaluation of the five causal consistency algorithms. We began by characterizing the system in which we ran the simulations and the default system settings. Next, we provided a characterization of the workloads and defined two novel features to help in this task, namely the Update Generation Rate Asymmetry (*GRA*) and the Object Ownership to Objects in Causal Past Ratio (*OPR*). Before we provided some experimental results we identified the potential costs and benefits that were tracked in our simulations. Finally, we presented the results of the empirical study and identified how each workload affects each algorithm. Among other results, we give special attention to the asymmetry of the system's load that appears to have the biggest impact on the algorithms' performance.



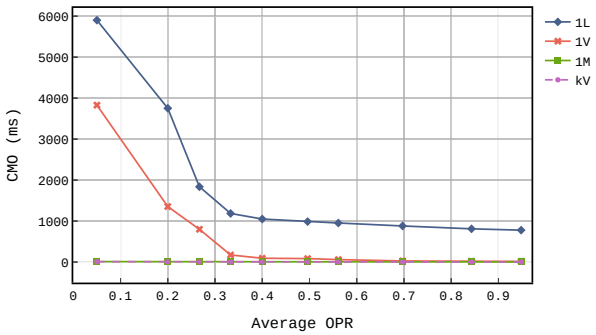
(a) *GRA* $= 0$



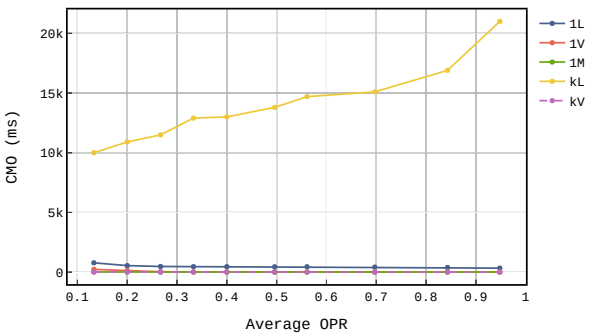
(b) *GRA* $= 0.5$



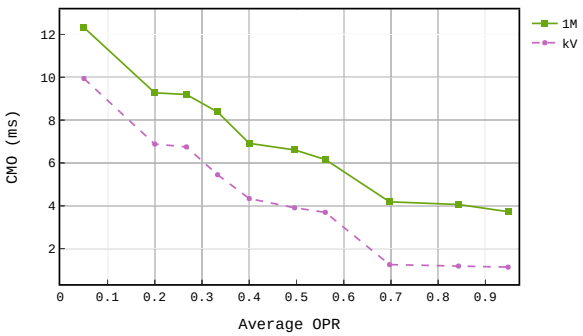
(c) *GRA* $= 0.7$



(d) *GRA* $= 1$

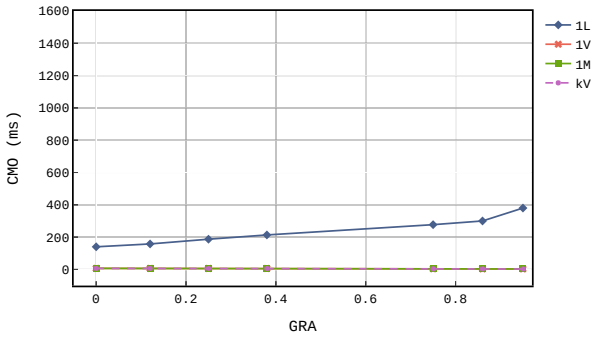


(e) *GRA* $= 0.7$; kL included

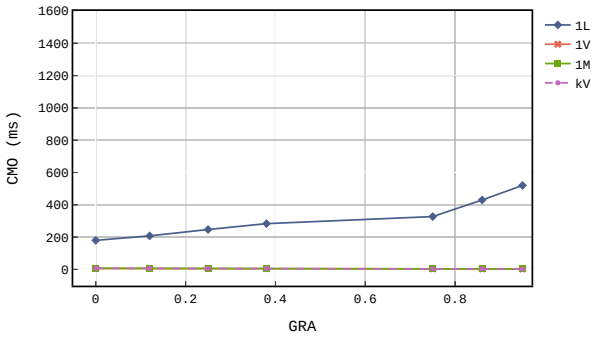


(f) *GRA* $= 0.7$; 1M and kV experiments only

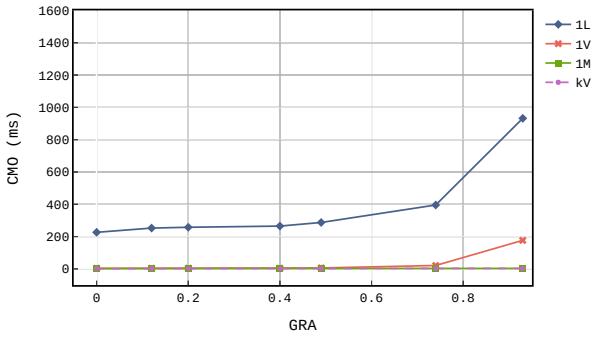
Figure 5.2: Variable *OPR* scenarios: CMO 95th percentile for varying *OPR* values;$N = 16; K = 1600; J_i = 45ms$; multiple *GRAs* considered; variable uniform R(k); zipfian P(K)



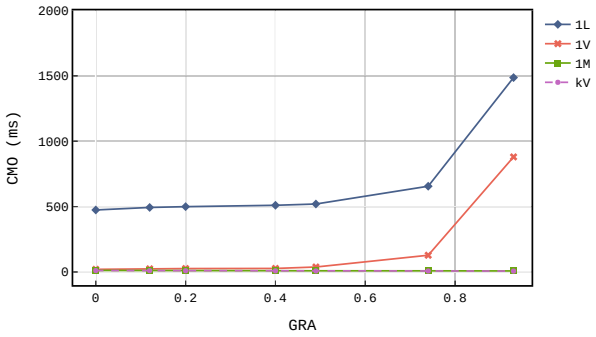
(a) *OPR* $= 1;\ J_i = 0ms; R(k) = N$ uniform $P(k)$



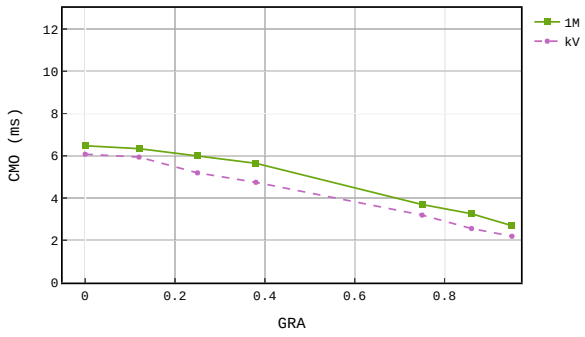
(b) *OPR* $= 1;\ J_i = 50ms; R(k) = N$ uniform $P(k)$



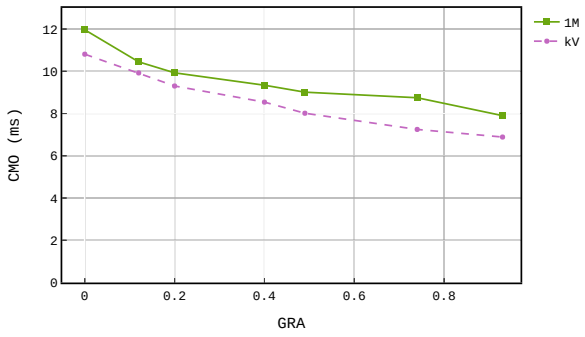
(c) *OPR* $= 0.26;\ J_i = 0ms;$; zipfian $P(k)$



(d) *OPR* $= 0.26;\ J_i = 50ms$; zipfian $P(k)$



(e) *OPR* $= 1;\ J_i = 0ms; R(k) = N$; 1M and kV included; uniform $P(k)$;



(f) *OPR* $= 0.26;\ J_i = 50ms$; 1M and kV included; zipfian P(k)

Figure 5.3: Variable GRA scenarios: CMO 95th percentile for varying *GRA* values;variable exponential T_i; $N = 16; K = 1600$; uniform R(K); multiple *OPRs* considered

6

Analysis and Takeaways

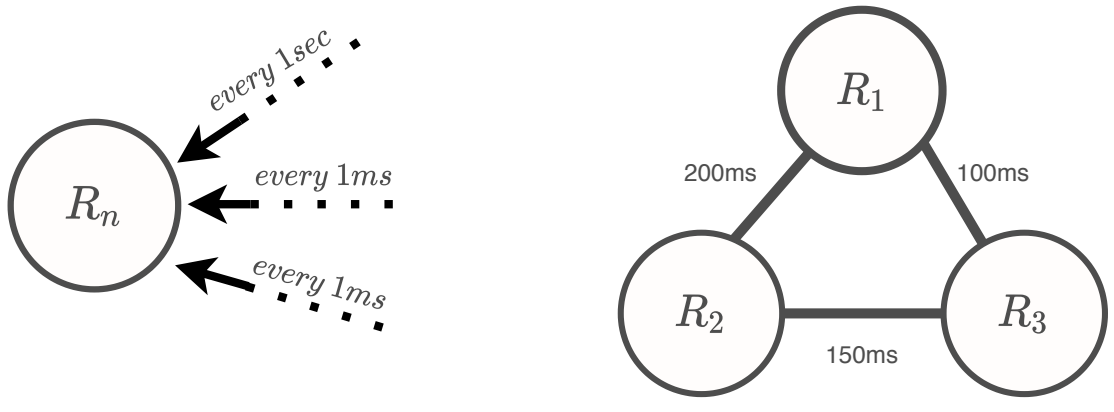
Contents

Our experiments highlight some interesting properties of the different metadata choices. This chapter enumerates the main takeaways from the empirical study and derives a decision chart that aims at helping system designers to pick the right technique for their applications, namely, Section 6.1 describes the metadata properties that originated from the empirical study, and in Section 6.2 the decision tree is presented.

6.1 Metadata Properties

6.1.1 1L only performs well in symmetric scenarios:

Lamport clocks cannot capture concurrency and do not allow us to identify the source of updates in the causal past of another update. This forces a node to wait for updates from every other node in the system in order to make a single update visible. From the point of view of a single node, it is intuitive that all updates will be delayed by the rate of the slowest node. Consider the simple example of Figure 6.1(a) where a node receives updates every 1 millisecond from every other node except for one that only sends updates at a rate of one update every second. The node can only apply part of all the previously received updates once every second because of a single node. It is then clear that when nodes produce updates at different paces, the entire system is affected by the rate of the slowest node in the system.



(a) **(b)**

Figure 6.1: Example of asymmetry in the rate of updates and network in a system

6.1.2 Even in symmetric scenarios, 1L is affected by the network latency:

Picking up from the previous point, if we assume that the system is symmetric i.e. nodes produce updates at exactly the same pace. In order to deliver an update u with logical clock x, node i needs to receive an update with timestamp $\geq x$ from every other node. Even if these updates have been produced exactly at the same real-time instant as update u there is most likely some asymmetry that is introduced by the underlying network latency. Consider the example of Figure 6.1(b) In this scenario, if replica R2 and R3 produced an update at the same time, replica R1 would only see the update generated by replica R2, 100ms after it had received the update from replica R3. From our experiences, the additional delay introduced by the network appears to tend towards the average network latency. This is clearly visible in Figure 5.4. This suggests that Lamport clocks are not advisable in geo-replicated scenarios, where inter-node latencies are large and diverse.

6.1.3 kL brings no advantages w.r.t. 1L, expect in cases of extreme symmetry or extreme partial replication:

Although kL keeps additional detail over 1L, which is substantially more expensive, it brings little or no advantages to most scenarios. In fact, to use kL can even be detrimental to the performance of the system. This happens because the *CMO* of an update depends on both the GRA for that object and the *OPR*. Objects that are seldom accessed will have an extremely high *CMO*. Whenever they are updated, they will delay any updates that read that object.

To better understand this phenomenon consider the example of Figure 6.2. In this example we have three replicas R_1, R_2, and R_3 which together replicate objects A, B, and C. Notice that object B is fully replicated and replicas R_2 and R_3 both replicate object C. Depending on the access patterns of the clients, the kL algorithm will suffer a penalty in performance. Following the example, in replica R_1, a client generates updates to object B every one millisecond which are then propagated towards replica R_2 and R_3 (both also replicate object B). Once the update arrives at replica R_2, it also needs to wait for an update from replica R_3 to object B. Since object B is seldom accessed in replica R_3 (in our example it is only accessed every 10 seconds), it will take a long time before the pending update can be applied.

Notice that in replica R_3 there is a client that generates updates at the same rate as the client in replica R_1, however, these updates are towards object C which means that it will not advance forward the logical clock of object B, and therefore cannot unblock the queued updates in replica R_2. Compared to the 1L algorithm, since a single Lamport clock cannot capture which object was accessed, the updates made in replica R_3 towards object C are able to unblock the queued messages in replica R_2.

This phenomenon is magnified by the number of objects in the system, the level of replication of each object, the popularity of each object, and the number of dependencies between updates. For example,

if a client reads other objects (i.e., create a causal dependency) that suffer from the same problem as object B, then we would only be able to apply the update once all dependencies have been also been seen, which may take an indefinite time. This creates a cascading effect of causal dependencies that highly impacts the CMO of each replica.

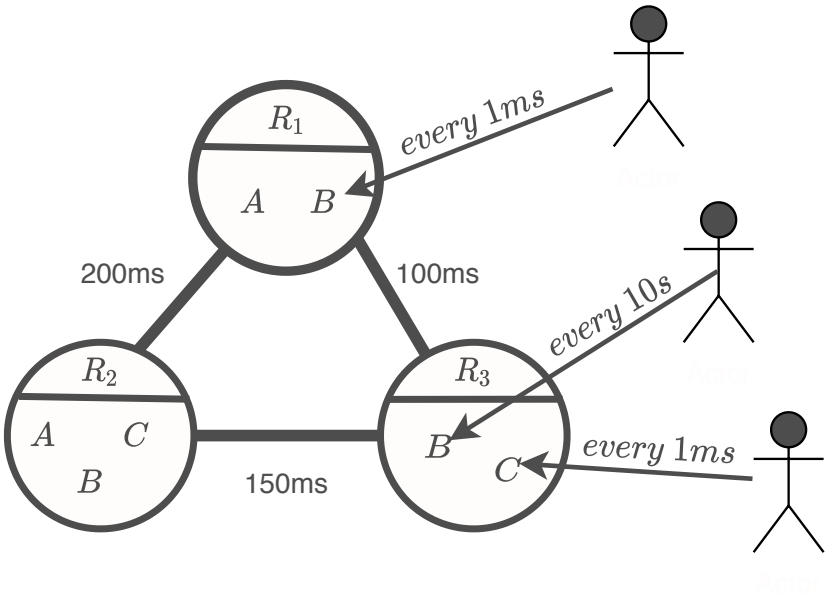


Figure 6.2: Example of a possible object placement and client's access patterns that results in poor performance when using the kL algorithm

There are, however, two exceptions to this. The first exception is when we have a uniform $R(x) = 2$, which, in that case, a certain node doesn't need to wait for all other nodes to perform an update on the said object; this results in the kL algorithm to perform similarly to kV. The other corner case is when all objects are updated at the same rate by all nodes; in such a scenario, kL would perform similarly to 1L. This result is somewhat unintuitive but also an interesting one because it means that having more metadata does not necessarily imply better performance. Apart from having more metadata, it is also necessary to use the metadata in the correct way for it to be useful. We can say that the kL algorithm suffers from a curse of knowledge.

6.1.4 1M and kV have similar performance:

Both 1M and kV have similar performance in our experiments. Thus, the choice between 1M and kV only depends on the size of the metadata. For scenarios where the number of objects is larger than the number of nodes, the 1M is more advisable. For scenarios where the number of objects is smaller than the number of nodes, kV is better. Note that, in many practical systems, it is possible to group objects in a few numbers of *partitions*, where objects in the same partition use the same replication strategy. In

this case, the use of kV can be a sensible choice since the size of the metadata would be considerably reduced.

6.1.5 Despite 1M and kV having similar performances, the kV algorithm can perform slightly better:

The kV algorithm establishes one FIFO connection per object. When the system has enough network bandwidth, it is intuitive that propagating messages in parallel results in a higher level of concurrency. When we compare multiple connections with having only one FIFO connection such as in 1M, there seems to exist room for some performance gain. Despite having this apparent benefit, the differences in the *CMO* are actually very low, which, in some cases, doesn't seem to pay up the much metadata cost. An example where this phenomenon would be amplified would be in a scenario where all updates to different objects were concurrent (i.e., clients did not generate causal dependencies by reading other objects).

6.1.6 1M and kV perform better in systems with higher GRA:

The 1M and kV experience a fairly lower number of false dependencies than other algorithms which are sufficient for the algorithms to output lower *CMO* values. However, there are other features that, allied to the reduced number of false dependencies, further increase their performance. These features are the asymmetry in the update generation rate and the degree of replication of objects. First, when updates are generated at different rates, it means that updates will also be propagated at different times. Secondly, when objects are fully replicated, all replicas will be interested in all updates. This results in fewer cases that generate false dependencies compared to partial replication. The first observation lets us conclude that the higher the GRA, the less will FIFO connections be saturated which means that, on average, updates arrive faster (the same applies to the dependencies of an update). This result allied with fewer false dependencies leads to smaller *CMO* values. This phenomenon can be observed in Figures 5.2(f), 5.3(e) and 5.3(f).

6.1.7 1M/kV significantly outperform 1V in systems where *OPR* is low:

In partially replicated systems, nodes do not store a replica of every object. Therefore, it is likely that in the causal past of updates that need to be applied at node i there are updates to objects that are not replicated by i (a false dependency). Keeping different vector clocks for each object, or a matrix clock offers the necessary detail to prevent nodes to wait for updates they will never receive. This is clearly visible in Fig. 5.2, where the 1V algorithm gains performance as the *OPR* increases.

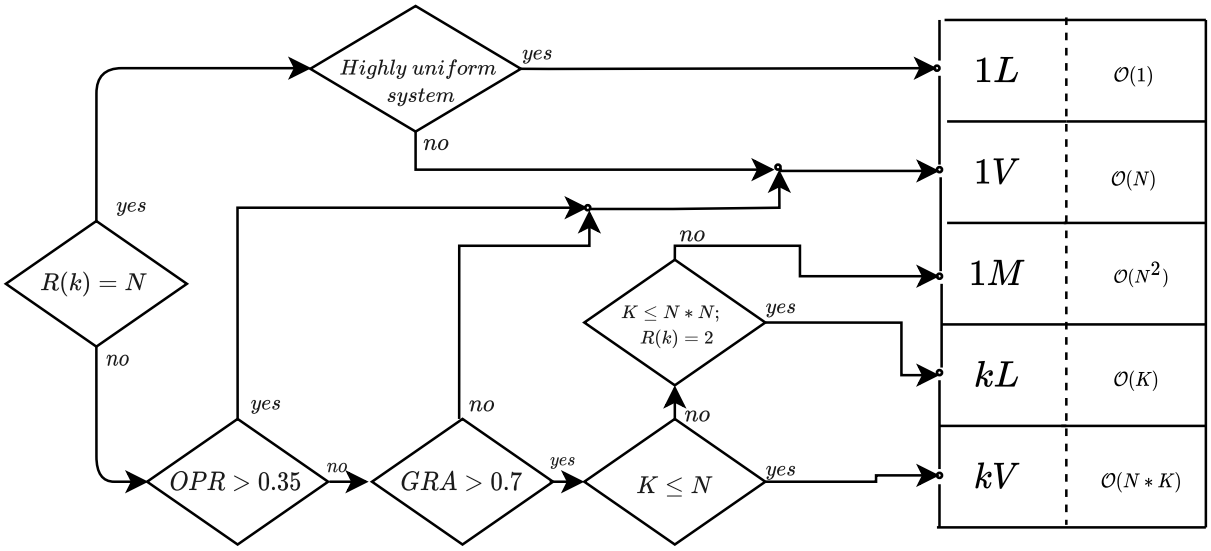


Figure 6.3: Decision chart for the various metadata schemes

6.2 Decision tree

Based on these insights, it is possible to construct a decision tree to help system designers select the most suitable metadata scheme for their storage system, as a function of the deployment and workload characteristics. The decision tree is depicted in Fig. 6.3. First, if the system is fully replicated, there is no need to use additional metadata and the choice is between the 1L and the 1V algorithms. We do not need to use extra metadata because as we are in a full replication scenario, all updates will reach all servers and, therefore, a single scalar is sufficient to track all necessary causal dependencies. In this scenario, if the system is not highly uniform (we consider a highly uniform system to be $OPR = 1;\ GRA = 0;\ J_i = 0;$ *uniform* $T_i;$ *uniform* $P(K);\ sd(\delta_{ij}) = 0$, where sd is the standard deviation), the wise choice is to use a vector clock, otherwise, if the system designer supports having higher *CMOs* (e.g. bandwidth/processing power may be scarce which justifies the use of less metadata at the expense of a higher *CMO*) then it is better to use a Lamport clock. If we are instead on a partially replicated scenario, we first check if our *OPR* is greater than 0.35. As we have seen, the higher the *OPR*, the better the 1V algorithm will perform. If the *OPR* is lower than 0.35, it means that most likely we also have few replicas for each object; In such a case, we must assess whether the system is uniform. From our previous analysis, we can correlate GRA with the system asymmetry. For *GRA* > 0.7 the 1V algorithm starts showing performance problems, therefore we should use a vector only when the *GRA* is lower than 0.7. We use the kV algorithm if the scenario where the number of objects is smaller than the number of nodes; otherwise, it is generally better to use a Matrix clock for the entire system. There is still a special case where objects are replicated in only two servers and the number of objects is smaller than N which in that case you should use kL.

6.3 Summary

In this chapter, we presented some of the takeaways that can be extracted from the empirical study of Chapter 5. More specifically, we identified some of the properties of the different metadata structures and were able to define in which scenarios a certain data structure is more suitable. We concluded that for most scenarios using a single vector clock is sufficient, however, its performance starts declining for less uniform systems. From these results, we derived a decision tree that can help system designers select the most suitable metadata scheme for their storage system.

7

Conclusions and Future Work

Contents

7.1 Conclusions

Ensuring causal consistency is at odds with the mechanisms used to keep track of causal dependencies. On the one hand, one can opt to use more detail to better track causality. On the other hand, this results in consuming a substantially larger amount of network bandwidth and storage space. In this paper, we addressed the problem of whether the benefits of using more complex structures are worth their cost in the context of partially replicated systems. We have shown that for some workloads the use of more expensive clocks does bring significant benefits and that for other workloads no visible benefits can be observed. To help system designers to pick the right mechanisms for their applications, the paper introduces two novel features, GRA and OPR, that capture relevant properties that affect the performance of different causality tracking mechanisms. Based on these features and an extensive experimental evaluation, we derived a decision chart that characterizes different scenarios where each type of clock is more beneficial.

7.2 Future Work

The analysis focused on three classic causal dependency tracking data structures. There is possibly some work that can be done to include other existing data structures in this study. An example of a possible data structure is causal histories which are extremely accurate in capturing causal dependencies at the cost of possibly having higher metadata sizes. Additionally, the work done in this thesis does not consider hybrid logical clocks which seem to be a very present approach in the industry. The assessment of their trade-offs compared to regular logical clocks is, however, not trivial.

Another possible line of work is based on using the presented results and methodology as a baseline for a possible system. One could idealize a system where the type of metadata structure used is defined at runtime and adapts to the current system workload. Additionally, different servers could communicate using different, but compatible, data structures as a way to minimize the costs of the system.

Bibliography

[1] M. Chandy and L. Lamport, “Distributed snapshots: Determining global states of distributed systems,” *ACM TOCS*, vol. 3, no. 1, p. 63–75, Feb. 1985.

[2] G. Ricart and A. Agrawala, “An optimal algorithm for mutual exclusion in computer networks,” *Commun. ACM*, vol. 24, no. 1, p. 9–17, Jan. 1981.

[3] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto, “Causal memory: definitions, implementation, and programming,” *Distributed Computing*, vol. 9, no. 1, pp. 37–49, Mar 1995.

[4] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[5] D. Parker, G. Popek, and et. al., “Detection of mutual inconsistency in distributed systems,” *IEEE Transactions on Software Engineering*, vol. 3, pp. 240–247, 1983.

[6] F. Mattern, “Virtual time and global states of distributed systems,” in *Proc. Workshop on Parallel and Distributed Algorithms*, 1989, pp. 215–226.

[7] S. Sarin and N. Lynch, “Discarding obsolete information in a replicated database system,” *IEEE Transactions on Software Engineering*, vol. 1, pp. 39–47, 1987.

[8] H. Guerreiro, L. Rodrigues, N. Preguiça, and N. Quental, “Causality tracking trade-offs for distributed storage,” *IEEE International Symposium on Network Computing and Applications*, 2020.

[9] R. Nelson, D. McCarthy, S. Malys, J. Levine, B. Guinot, H. Fliegel, R. Beard, and T. Bartholomew, “The leap second: its history and possible future,” *Metrologia*, vol. 38, no. 6, p. 509, 2001.

[10] A. Acharya and B. Badrinath, “Recording distributed snopshots based on causal order of message delivery,” *Information Processing Letters*, vol. 44, no. 6, pp. 317–321, 1992.

[11] M. Bravo, “Metadata management in causally consistent systems,” Ph.D. dissertation, Université Catholique de Louvain and Instituto Superior Técnico, Universidade de Lisboa, 2018.

[12] C. Fidge, “Logical time in distributed computing systems,” *IEEE Computer*, vol. 24, no. 8, pp. 28–33, 1991.

[13] R. Schwarz and F. Mattern, “Detecting causal relationships in distributed computations: In search of the holy grail,” *Distributed computing*, vol. 7, no. 3, pp. 149–174, 1994.

[14] B. Kang, R. Wilensky, and J. Kubiatowicz, “The hash history approach for reconciling mutual inconsistency,” in *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.* IEEE, 2003, pp. 670–677.

[15] P. Almeida, C. Baquero, and V. Fonte, “Version stamps-decentralized version vectors,” in *Proceedings 22nd International Conference on Distributed Computing Systems*. IEEE, 2002, pp. 544–551.

[16] ——, “Interval tree clocks,” in *International Conference On Principles Of Distributed Systems*. Springer, 2008, pp. 259–274.

[17] J. Almeida, P. Almeida, and C. Baquero, “Bounded version vectors,” in *International Symposium on Distributed Computing*. Springer, 2004, pp. 102–116.

[18] N. Preguiça, C. Baquero, P. Almeida, V. Fonte, and R. Gonçalves, “Dotted version vectors: Logical clocks for optimistic replication,” *arXiv preprint arXiv:1011.5808*, 2010.

[19] S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, “Logical physical clocks,” in *International Conference on Principles of Distributed Systems*. Springer, 2014, pp. 17–32.

[20] P. Mahajan, L. Alvisi, M. Dahlin *et al.*, “Consistency, availability, and convergence,” *University of Texas at Austin Tech Report*, vol. 11, p. 158, 2011.

[21] H. Attiya, F. Ellen, and A. Morrison, “Limitations of highly-available eventually-consistent data stores,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 141–155, 2016.

[22] C. Papadimitriou, “Serializability of concurrent database updates,” Massachusetts inst of tech cambridge lab for computer science, Tech. Rep., 1979.

[23] M. Herlihy and J. Wing, “Linearizability: A Correctness Condition for Concurrent Objects,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[24] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

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

[26] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400.

[27] E. Gokkoca, M. Altinel, R. Cingil, E. Tatbul, P. Koksal, and A. Dogac, "Design and implementation of a distributed workflow enactment service," in *Proceedings of CoopIS 97: 2nd IFCIS Conference on Cooperative Information Systems*. IEEE, 1997, pp. 89–98.

[28] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 11.

[29] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 360–391, 1992.

[30] S. Almeida, J. Leitão, and L. Rodrigues, "Chainreaction: a causal+ consistent datastore based on chain replication," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 85–98.

[31] D. Akkoorath, A. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2016, pp. 405–414.

[32] D. Didona, K. Spirovska, and W. Zwaenepoel, "Okapi: Causally consistent geo-replication made faster, cheaper and more available," *arXiv preprint arXiv:1702.04263*, 2017.

[33] C. Gunawardhana, M. Bravo, and L. Rodrigues, "Unobtrusive deferred update stabilization for efficient geo-replication," in *USENIX Annual Technical Conference (ATC)*, 2017, pp. 83–95.

[34] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–13.

[35] K. Spirovska, D. Didona, and W. Zwaenepoel, "Optimistic causal consistency for geo-replicated key-value stores," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2626–2629.

[36] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *2010 29th IEEE Symposium on Reliable Distributed Systems*. IEEE, 2010, pp. 214–224.

[37] M. Bravo, L. Rodrigues, and P. Van Roy, "Saturn: A distributed metadata service for causal consistency," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 111–126.

[38] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, "Practi replication." in *NSDI*, vol. 6, 2006, pp. 5–5.

[39] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch, "The bayou architecture: Support for data sharing among mobile users," in *1994 First Workshop on Mobile Computing Systems and Applications*. IEEE, 1994, pp. 2–7.

[40] P. Fouto, J. Leitão, and N. Preguiça, "Practical and fast causal consistent partial geo-replication," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2018, pp. 1–10.

[41] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro, "Write fast, read in the past: Causal consistency for client-side applications," in *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015, pp. 75–87.

[42] P. Bailis, A. Fekete, A. Ghodsi, J. Hellerstein, and I. Stoica, "The potential dangers of causal consistency and an explicit solution," in *Proc. SOCC*, Oct. 2012, pp. 1–7.

[43] K. Spirovska, D. Didona, and W. Zwaenepoel, "Paris: Causally consistent transactions with non-blocking reads and partial replication," *arXiv preprint arXiv:1902.09327*, 2019.

[44] Z. Xiang and N. Vaidya, "Global stabilization for causally consistent partial replication," *arXiv preprint arXiv:1803.05575*, 2018.

[45] T. Crain and M. Shapiro, "Designing a causally consistent protocol for geo-distributed partial replication," in *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, 2015, p. 6.

[46] A. van der Linde, P. Fouto, J. Leitão, N. Preguiça, S. Castiñeira, and A. Bieniusa, "Legion: Enriching internet services with peer-to-peer interactions," in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 283–292.

[47] T. Mahmood, S. Narayanan, S. Rao, T. Vijaykumar, and M. Thottethodi, "Karma: Cost-effective geo-replicated cloud storage with dynamic enforcement of causal consistency," *IEEE Transactions on Cloud Computing*, 2018.

[48] D. Cheriton and D. Skeen, "Understanding the limitations of causally and totally ordered communication," in *Proc. SOSP*, 1993, pp. 44–57.

[49] M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. Rodrigues, "On the use of clocks to enforce consistency in the cloud." *IEEE Data Eng. Bull.*, vol. 38, no. 1, pp. 18–31, 2015.

[50] K. Birman, A. Schiper, and P. Stephenson, “Lightweight causal and atomic group multicast,” *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 3, p. 272–314, Aug. 1991.

[51] A. Montresor and M. Jelasity, “Peersim: A scalable P2P simulator,” in *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*. IEEE, 2009, pp. 99–100.

[52] JDK 14, “openjdk,” https://openjdk.java.net/projects/jdk/14/, [Online; accessed 14-October-2020].

[53] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “A comprehensive study of Convergent and Commutative Replicated Data Types,” Inria – Centre Paris-Rocquencourt ; INRIA, Research Report RR-7506, Jan. 2011. [Online]. Available: https://hal.inria.fr/inria-00555588

[54] “Home: Cloudping - aws latency monitoring.” [Online]. Available: https://www.cloudping.co/grid

[55] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, “Web caching and zipf-like distributions: Evidence and implications,” in *IEEE INFOCOM*, vol. 1, 1999, pp. 126–134.