TÉCNICO LISBOA

Mechanisms for Providing Causal Consistency on Edge Computing

Nuno Cerqueira Afonso

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. Luís Eduardo Teixeira Rodrigues
Member of the Committee: Prof. Nuno Manuel Ribeiro Preguiça

November 2018

Acknowledgments

Firstly, I would like to thank Professor Luís Rodrigues for accepting me as his student and guiding me throughout this journey. I thank Manuel Bravo for his tireless help, by always being available for discussing the progress and coming up with important ideas. I thank my family for being an endless source of support and a safety net when things do not go as planned. Last but not least, I thank my friends who I have come to know during the years. I know I can rely on you, no matter what.

To each and every one of you – Thank you.

Abstract

Today, many applications offload computation and storage to the cloud. Unfortunately, the high network latency between clients and datacenters can impair novel, latency-constrained applications, such as augmented reality, real-time image processing and collaborative applications. Edge computing has emerged as a potential solution to circumvent this problem. To unleash its full potential, the edge must cache data that is frequently used. However, building a storage service that is able to maintain many (partial) replicas while providing meaningful consistency guarantees is an open challenge. In this thesis, there is a presentation of Gesto, a data storage architecture that enables scalable causal consistency for edge networks. Gesto integrates a novel causality tracking mechanism that relies on multipart timestamps of constant size, independently on the number of edge caches. As evaluation shows, this mechanism enables Gesto to simultaneously offer low local operation latency, scalability, high throughput, fast update replication, and, unlike previous work, quick client migrations.

Keywords

Causal Consistency; Cloudlet; Datacenter; Edge Computing; Metadata.

Resumo

Atualmente, muitas aplicações delegam a computação e o armazenamento na nuvem. Infelizmente, a alta latência da rede entre os clientes e os centros de dados pode prejudicar novas aplicações com determinadas restrições. Realidade aumentada, processamento de imagem em tempo-real e aplicações colaborativas são alguns exemplos. A computação na periferia tem emergido como uma potencial solução para contornar este problema. De forma a atingir o seu máximo potencial, os dispositivos da periferia devem servir os dados que são utilizados mais frequentemente. No entanto, a construção de um serviço de armazenamento que é capaz de manter tantas réplicas (parciais) enquanto oferece consistência significativa é ainda um problema sem solução. Nesta tese, apresenta-se o Gesto, uma arquitetura de armazenamento de dados que possibilita um crescimento da consistência causal até às redes da periferia. Gesto integra uma nova técnica para rastrear a causalidade, que recorre a estampilhas multi-parte de tamanho constante, independentemente do número de réplicas. Como a avaliação mostrará, este mecanismo faz com que Gesto forneça, em simultâneo, baixas latências de operações na réplica local, crescimento, elevado débito das operações, rápida replicação de atualizações e, contrariamente a trabalho anterior, ótimas migrações de clientes.

Palavras Chave

Consistência Causal; Micro Centro de Dados; Centro de Dados; Computação na Periferia; Metadados.

Contents

1 Introduction **2**

1.1 Motivation 3

1.2 Contributions 4

1.3 Results 5

1.4 Research History 5

1.5 Structure of the Document 5

2 Related Work **6**

2.1 Edge Computing 7

2.1.1 Challenges and Requirements 7

2.1.2 Edge Storage Systems 8

2.2 Causal Dependencies 9

2.3 Metadata Management 11

2.3.1 Timestamp Part Encoding 11

2.3.2 Timestamp Size 11

2.3.3 Timestamp Assignment 15

2.4 Causally Consistent Systems 15

2.4.1 Kronos [1] 16

2.4.2 COPS [2] 16

2.4.3 Orbe [3] 17

2.4.4 PRACTI [4] 18

2.4.5 Lazy Replication [5] 20

2.4.6 ChainReaction [6] 21

2.4.7 SwiftCloud [7] 22

2.4.8 Legion [8] 23

2.4.9 Cure [9] 24

2.4.10 Okapi [10] 25

2.4.11 Eunomia [11] 26

2.4.12 GentleRain [12] . . . 27
2.4.13 POCC [13] . . . 28
2.4.14 Occult [14] . . . 29
2.4.15 Saturn [15] . . . 30
2.4.16 C^3 [16] . . . 31
2.4.17 Comparison . . . 32
2.5 Limitations of Cloud Solutions on the Edge . . . 35

3 Gesto: Geo-referenced Edge Store **37**
3.1 Goals . . . 38
3.2 Design . . . 38
3.2.1 System Components . . . 38
3.2.2 Client Interaction . . . 39
3.3 Multipart Timestamps . . . 41
3.4 Protocols . . . 42
3.4.1 Reads and Writes . . . 43
3.4.2 Intra-regional Update Replication . . . 43
3.4.3 Client Migration . . . 44
3.4.4 Inter-regional Update Replication . . . 47
3.5 Correctness . . . 48
3.6 Reconfiguration and Faults . . . 50
3.6.1 Directory Service . . . 50
3.6.2 Changing the Replication Set . . . 51
3.6.3 Adding and Removing Instances . . . 52
3.6.4 Faults . . . 52

4 Implementation **54**
4.1 Framework for Edge Storage . . . 55
4.1.1 Client . . . 55
4.1.2 Client Receiver . . . 56
4.1.3 Partition . . . 56
4.1.4 Timestamp Sender . . . 57
4.1.5 Timestamp Receiver . . . 57
4.1.6 Broker . . . 58
4.2 Implementing Different Systems . . . 58
4.2.1 Eventual Consistency . . . 58
4.2.2 COPS . . . 59

4.2.3 Occult 60
4.2.4 Gesto 60
4.2.5 Saturn 61

5 Evaluation **62**
5.1 Goal 63
5.2 Experimental Settings 63
5.3 Local Operation Latency 65
5.4 Throughput 65
5.5 Remote Update Visibility 66
5.6 Scalability 68
5.7 Migration Latency 69
5.8 Discussion 70

6 Conclusion **72**
6.1 Conclusions 73
6.2 System Limitations and Future Work 73

List of Figures

2.1 Run for demonstrating the causal consistency rules. 9
2.2 Run with concurrent operations. 10
2.3 Causal consistency with timestamps with one part per server. 13
2.4 Causal consistency with timestamps with one part for the entire system. 14
2.5 Graphic distribution of the systems according to their false dependencies and metadata size. Δ stands for genuine partial replication and Δ^* is for non-genuine. Low false dependencies stand for only one level on the table. Medium is either two or three. High is all of them. 34

3.1 Gesto's three-layer architecture. The numbers represent the steps that an update issued at the leftmost Gesto instance of *region 1* will follow until being propagated to the rightmost Gesto instance of *region 2*. The execution flow does not depict the payload transmission within regions (done by 5). Steps with ■ indicate that they require the update's payload to be executed. 39

4.1 Interaction of the different components for systems that propagate the payload alongside the metadata. 58
4.2 Interaction of the different components for systems that propagate the payload separately from the metadata. 59

5.1 Each system's read and write latencies. 64
5.2 Each system's throughput for a given access pattern, while maintaining the same read-/write ratio. 66
5.3 Each system's 90th percentile of remote update visibility for a given access pattern, while maintaining the same read/write ratio. 67
5.4 Cumulative distribution function for each system's remote update visibility. 68
5.5 Each system's throughput for a given number of clients. 69
5.6 Cumulative distribution function for each system's migration latency. 70

List of Tables

2.1 Comparison of the different solutions. K is the number of all objects in the system. N is the number of partitions. M is the number of replicas and R is the same, but per container. E, S, ID and ED represent inter-datacenter element, intra-server, intra-datacenter and inter-datacenter dependencies, respectively. $^{\checkmark}$ and $^{\times}$ stand for genuine and non-genuine partial replication, respectively. Regarding SwiftCloud's partial replication, there is a client cache that only replicates its interest set, but the datacenters have a full copy. 33

5.1 Average latencies (half RTT) in milliseconds among locations: Lille (Li), Luxembourg (Lu), Lyon (Ly), Nancy (Nc), Nantes (Nt), Rennes (R) and Sophia (S). 64

5.2 High level comparison of the evaluation results for each causally consistent system. . . . 70

List of Algorithms

1 Client code. 42

2 Gesto instance code (*cloudlet*). 45

3 Regional broker code (*broker*). 46

Acronyms

CDF Cumulative Distribution Function

CID Client Identifier

COPS Cluster of Order-Preserving Servers

CRDT Conflict-free Replicated Data Type

FIFO First In, First Out

IoT Internet of Things

LRU Least Recently Used

lts Local Timestamp

MEC Mobile Edge Computing

NTP Network Time Protocol

POCC Physical clock Optimistic Causal Consistency

rts Regional Timestamp

UID Unique Identifier

1

Introduction

Contents

1.1 Motivation 3
1.2 Contributions 4
1.3 Results 5
1.4 Research History 5
1.5 Structure of the Document 5

This thesis addresses the problem of implementing a causally consistent data store that can use many small datacenters, known as *cloudlets*, which are closely located to the edge of the network. Therefore, they are near clients and ensure that data access has low latency. These cloudlets complement the service provided by the current cloud services, in order to support applications that require low latency, such as: augmented reality, virtual reality, and collaborative work. Causally consistent data stores designed for the cloud fail to work on such setting, given that their design is based on assuming a relatively small number of replicas, full replication, and a stable network. Such assumptions do not hold for the edge, because: cloudlets have limited capabilities, specially in terms of processing power and storage; the number of replicas must be orders of magnitude greater, to cope with the increasing number of devices and users; the network suffers from much more frequent changes, which may result in clients to be forced to move to other replicas. This thesis presents a novel design that takes these new constraints into consideration, to offer causal consistency with low latency at the edge of the network.

1.1 Motivation

Cloud computing is an established paradigm and most applications, including those that run on mobile devices, use the cloud to fetch data, upload information, or offload computations that are resource intensive and can drain the battery of the devices [17]. Image processing for face or object recognition [18,19] is an example of a resource-eager task that can be offloaded to the cloud. This functionality is key for many augmented reality applications such as marker detection [18] or just-in-time video indexing [20]. However, a response time below 5–30 milliseconds is typically required for these applications to be usable [21]. This may be impossible to guarantee when accessing remote datacenters.

To circumvent this limitation, the use of computing resources closer to the logical extreme of the network, such that the cloud is pushed closer to clients, has emerged as a viable solution. Many approaches based on this principle are being advocated, including the use of *cloudlets* [22], *Mobile Edge Computing (MEC)* [23], and *fog computing* [24]. In addition to removing the latency bottleneck, edge computing brings other advantages, such as decreasing the load on networks and datacenters [25], and making the applications more robust to datacenter outages [21].

To unleash their full potential, edge nodes should not only provide processing capacity, but also cache data that may be frequently used [26]; otherwise, the advantages of processing on the edge may be impaired by frequent remote data accesses [27]. By using cached data, end-users' requests rarely need to be served by the root datacenters. Thus, even applications without tight latency constraints can benefit from edge computing. For instance, Snapchat [28] generates more than 3.5 billion publications each day [29]. These are deleted after 24 hours and are mostly seen by close friends and family [30–32], who are geographically near to the user [33]. In this context, edge storage can offer a better user

experience and save important network resources by avoiding the upload of ephemeral data to the cloud.

Consequently, a key ingredient of edge assisted cloud computing is a storage service that extends the one offered by the cloud, in a way that relevant data is replicated closer to the edge. Nevertheless, designing such a storage service while providing meaningful consistency guarantees is not trivial. Specifically, the problem of supporting causal consistency on the edge is particularly interesting, given that causal consistency was shown to be the strongest consistency criteria that can be offered without compromising availability [34]. The edge scenario enforces a set requirements that make the implementation of causal consistency hard:

High Scalability. An important observation is that, to ensure acceptable latency, many edge nodes will need to be deployed. For instance, to ensure the target $5-30$ milliseconds latency, every heavily populated region of $95\ 114$ Km^2 (a circle of 174 Km radius) should have its own edge instance at the center. This number is based on measured round-trip-times (RTTs) among Amazon EC2 regions (RTT of 22.5 ms between the regions of Oregon and California, which are 783 Km apart), such that the 5 ms lower-bound is met. In Europe alone, with a total area of 10.18 million Km^2, approximately 107 replicas should be deployed.

Partial Replication. Edge replicas will only maintain partial information given that: (i) for efficiency, only information that is valuable to the clients that a replica is serving should be shipped and stored on the edge (*genuine partial replication*); (ii) edge nodes are resource-constrained.

Non-sticky Sessions. Clients should be able to efficiently move among replicas without having to restart their session. Clients may move, for instance, due to mobility or to read data that it is not replicated in their preferred replica.

Interestingly, most systems that offer causal consistency in the cloud make exactly the opposite assumptions [2–16]: they have been developed for scenarios where exist few datacenters, each datacenter usually implements full replication, and clients remain mostly connected to a single datacenter.

The goal of this thesis is to develop a system for enforcing causal consistency that fulfills all the previously enumerated requirements, while keeping competitive performance with the existing solutions for the cloud.

1.2 Contributions

This thesis compares, implements and evaluates strategies for enforcing causal consistency at the edge of the network. The resulting contributions are the following:

- Proposal of a new multi-layer architecture for bringing causally consistent data stores to the masses,

through the use of edge devices. It focuses on fast operations to the replica the clients are connected to and allows the unrestricted movement of clients by relying on a simple topology and small metadata. Because of the intrinsic characteristics of the network edge, there are reconfiguration algorithms that allow changes on the amount of system replicas and on their set of replicated objects.

- Detailed analysis and comparison (both theoretical and experimental) of the different techniques for ensuring causal consistency. Given the lack of work in applying causal consistency in this relatively new setting, there is a discussion on how some techniques cannot be successfully applied.

1.3 Results

This thesis produced the following results:

- An implementation of Gesto, which is a system that adapts particular strategies for enforcing causal consistency at the network edge.
- An experimental evaluation of the system implementation, regarding its performance. To better position Gesto in respect to other causally consistent strategies, some representative systems were included as a way of creating acceptable boundaries.

1.4 Research History

A paper that presents parts of this work has been published in [35], while also being included in the shortlist for the best paper award.

This work was developed at INESC-ID and was partially funded by Fundação para a Ciência e Tecnologia (FCT) as part of the projects PTDC/EEI-COM/29271/2017 (Cosmos) and UID/CEC/50021/2013.

1.5 Structure of the Document

This thesis is is organized as follows. Chapter 2 presents some applications developed for the edge and introduces causal consistency with some techniques that allow data stores with this consistency guarantee. Chapter 3 describes the design of Gesto. Chapter 4 addresses the implementation of Gesto's prototype, as well as the other systems considered for evaluation. Chapter 5 reveals the results of the evaluation and makes some remarks about the differences among the systems. Chapter 6 concludes this thesis by outlining the discoveries and unveils some directions for future work.

2

Related Work

Contents

2.1 Edge Computing 7
2.2 Causal Dependencies 9
2.3 Metadata Management 11
2.4 Causally Consistent Systems 15
2.5 Limitations of Cloud Solutions on the Edge 35

This chapter starts by introducing the concept of edge computing, its challenges and requirements, and some systems that provide storage at the network edge (Section 2.1). Then, there is a definition of causal consistency (Section 2.2) and an overview of some strategies for enforcing it (Section 2.3). Afterwards, there is a survey of some causally consistent systems, which ends with a summary of their main differences (Section 2.4). This chapter ends by addressing the limitations of the causally consistent solutions for the cloud, when applied to the network edge (Section 2.5).

2.1 Edge Computing

The term *edge computing* [36] captures a model of distributed computing that aims at leveraging the processing capacity that exists in the devices that operate on the network edge. Edge computing augments the cloud computing paradigm, where all processing is performed in datacenters, in order to achieve better scalability, lower latencies, better data privacy, and a more efficient usage of resources (including a more effective energy consumption). In edge computing, processing is performed cooperatively by edge devices and cloud servers. Which computations are performed on which of these components depend on a number of factors, including both the capacity of the nodes and the latency requirements.

Three possible approaches are: *cloudlets* [22], *MEC* [23], and *fog computing* [24]. In [37], the main differences are: cloudlets are computers or clusters of computers with above average performance and well-connected to the Internet; MEC is an architecture for deploying a wide variety of applications on top of some host and it is key for the growth of the *Internet of Things (IoT)*; fog computing needs both the cloud and the edge nodes (the others can exist without the cloud) and it runs applications for satisfying a specific use case.

In the next sections, there is an overview of the challenges and requirements that all these approaches share. Then, there are descriptions of some systems that are able to fulfill some of the requirements and provide a storage service at the network edge.

2.1.1 Challenges and Requirements

To materialize the potential of edge computing, a number of challenges need to be overcome [36]. The most obvious one is the heterogeneity of the machines involved in the computation, which may include a mixture of powerful and reliable nodes that run on datacenters and small, resource-constrained, unreliable nodes that run on the edge. Even among the edge nodes there is a large heterogeneity, since laptops and mobile phones are much more powerful than other IoT devices embedded in home appliances. An immediate consequence of this heterogeneity is that different devices have different capacities, and it is unrealistic to expect full replication of the data that an application manages.

Another challenge is that the number of devices is extremely large and, therefore, the size of control information required to enforce system guarantees, such as causal consistency, cannot be a function of the number of nodes. For instance, straightforward techniques such as relying on full vector clocks (described in Section 2.3) to keep track of causality are impractical.

Network conditions are also very heterogeneous and dynamic. Latencies among edge nodes vary widely and some parts of the topology can be very dynamic while others are mainly stable. Algorithms that operate on the edge must take this diversity into account and provide reconfiguration protocols when deemed necessary.

Finally, the edge network is subject to a phenomena known as *churn* [38], which consists of a frequent change in the operational status of edge devices. They can become connected and disconnected very frequently, or even fail permanently. Like in the previous challenge, there is the need for some additional measures to cope with these changes.

2.1.2 Edge Storage Systems

Usually, edge storage systems are developed for a specific application use (mostly recognition, as in [19, 39]) and not for general purpose. This is due to the high volumes of expected traffic and the limited bandwidth to the cloud infrastructure. Before transferring the data, there might be some pre-processing to either reduce the traffic or avoid it at all cost. Also, these storage systems choose lower consistency guarantees, in order to reduce the storage and latency overheads associated with stronger models.

One of the earliest systems for caching recognition applications is Cachier [19]. Most algorithms for recognizing samples work in a pipeline fashion and go through the following stages: *extract features*, *classify and match features* and *choose the best match*. They are compute intense, which does not allow them to run on the edge nodes. As a compromise, these edge nodes might store parts of the model, in order to use them to reply to similar requests. This approach grounds itself on the fact that clients around the same time and location request similar things with high probability. If a client requests something that cannot be fulfilled by the cached model, the request is forwarded to the cloud, which replies with the result and the component responsible for getting it. The edge node sends the response to the client and caches the new component, following an initial Least Recently Used (LRU) policy that is modified along the time. The main goal is to leverage a strategy that maximizes the amount of times the edge node is able to single-handedly reply to the client.

Other situation is when data must be archived, but not all of it is interesting. On [39], the system monitors a region through video capturing. Given that transferring all frames to the cloud requires huge network bandwidth and increases the analysis latency, edge nodes run an algorithm to create *feature vectors* that allow a relatively fast comparison between consecutive frames. If two frames have a similarity index above a predefined threshold, the differences are not significant and one of them can be

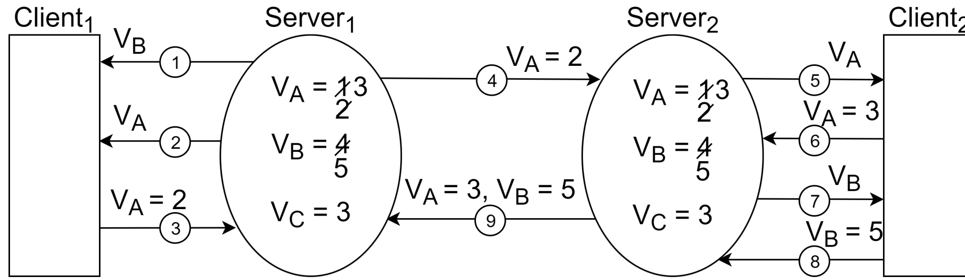

Figure 2.1: Run for demonstrating the causal consistency rules.

discard. Otherwise, the frame and its feature vector are timestamped and tagged with an unique node identifier. Then, all is put into a transmit buffer to be sent to the cloud.

CloudPath [27] provides a more general extension of eventually consistent storage systems to the network edge, while adding processing capabilities. It employs *path computing* to deploy a multi-level hierarchy, going from a datacenter to the edge node. As nodes get closer to end-users, their performance gradually decreases. Overall, CloudPath connects three types of nodes in a tree: *edge* nodes (the closest to end-users), *cloud* nodes (the most resourceful and the farthest away) and *core* nodes (any node between the last two). The root node replicates every object, while each descendent stores a subset of the items available at its parent. Clients read and write to their local replicas, which propagate the changes in the background, through the hierarchy. If a client tries to read data that is not locally available, the node redirects the request to its parent and, until the data is found, the request goes up through the nodes. When there is a response, it follows the opposite path, making each contacted child store the new data. This creates an on-demand data replication, as an attempt to only cache the latest requests. Updates are registered in a write log with a tag that contains the time of the change and the node identifier. As a fault tolerance measure, updates are marked as either *dirty* or *clear*. Dirty updates have not been acknowledged by the parent node, but clear ones are replicated on both.

2.2 Causal Dependencies

It is today well known that, in a distributed system subject to faults and network partitions, it is impossible to achieve simultaneously strong consistency, availability, and partition tolerance. This fact has been captured in the CAP theorem [40]. Therefore, there is the need to drop one of these desirable properties. A reasonable tradeoff is to weaken the consistency of the system, while still providing precise properties that can simplify the application design. Causal consistency has the interesting property that has been shown to be the strongest level of achievable consistency without blocking the system [34].

Causal consistency ensures that updates are observed in a order that is consistent with causality. Knowing exactly which updates are causally related requires knowledge about the application semantics.

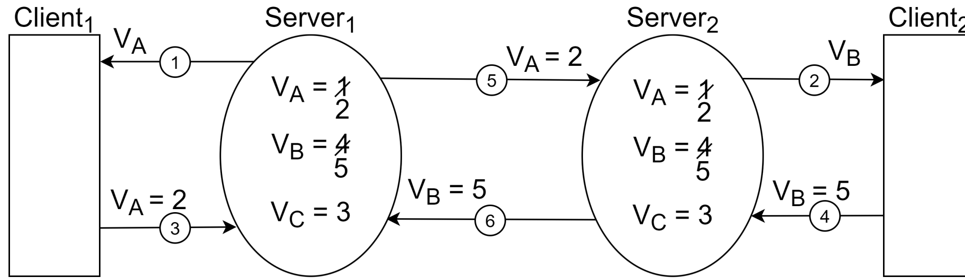

Figure 2.2: Run with concurrent operations.

Thus, most systems preserve the ordering among any updates that can *potentially* be causally related. The conditions for (potential) causal dependencies have been precisely identified by Lamport in [41]. Here, there is an illustration of these conditions with the help of Figure 2.1.

Firstly, all operations that are executed in serial order by the same thread are assumed to be causally dependent. In the figure, this is illustrated in the sequence of operations executed by Client$_1$: operation Op_3 is executed after Op_2, which in turn is executed after Op_1. This creates a dependency among these operations, denoted: $Op_1 \rightarrow Op_2 \rightarrow Op_3$. On Client$_2$, there is a similar scenario and thus: $Op_5 \rightarrow Op_6 \rightarrow Op_7 \rightarrow Op_8$.

Furthermore, in a shared memory system, causal dependencies can also be created when a thread reads an update that has been created by another thread. In the example, given that Client$_2$ reads the version of A written by Client$_1$, operation Op_5 depends on Op_3.

Finally, causality is transitive. Since $Op_2 \rightarrow Op_3$, $Op_3 \rightarrow Op_5$ and $Op_5 \rightarrow Op_6$, $Op_2 \rightarrow Op_6$ is still a valid dependency.

Sometimes, it is not possible to derive a causal dependency between two operations. In Figure 2.2, $Op_1 \nrightarrow Op_2$ and $Op_2 \nrightarrow Op_1$. Similarly, $Op_3 \nrightarrow Op_4$ and $Op_4 \nrightarrow Op_3$. Both pairs of operations are *concurrent* and they can be applied in any order.

To capture that fact that two operations are causally related, the system is required to maintain some metadata. Typically, this is achieved by timestamping each operation with some form of (logical, physical or hybrid) clock. In the following section, there is a description of the main techniques that have been used in the literature to perform such timestamping.

It is important to note that causal consistency, when used in settings where data is replicated, does not ensure that replicas have the exact same state, even if all updates are delivered to all replicas. This discrepancy is possible, because concurrent updates can be applied in different orders to different replicas. One way of avoiding it is to use data types that ensure convergence, regardless of the order by which concurrent updates are performed. They are known as Conflict-free Replicated Data Types (CRDTs). Systems that rely on CRDTs are SwiftCloud [7], Legion [8] and Cure [9]. Another alternative is to use some criteria to totally order concurrent updates, such as the *last-writer-wins rule*. In this

case, it is possible to ensure that replicas eventually converge to the same state, a property that has been named *causal+ consistency*. Examples of systems that offer this type of guarantees are Cluster of Order-Preserving Servers (COPS) [2], Orbe [3], ChainReaction [6], Okapi [10], Physical clock Optimistic Causal Consistency (POCC) [13] and C^3 [16].

2.3 Metadata Management

In order to ensure that updates can be applied in an order that respects causality, each update needs to be tagged with some amount of metadata. This metadata is called a *timestamp*. The timestamp encodes information regarding the causal past of the tagged operation. Since causal order is a partial order, each operation can have more than one predecessor in the causal graph. Therefore, to accurately capture the fact that the causal past of an operation can consist of multiple concurrent past operations, the timestamp may need to have multiple components, being called a *multipart timestamp*.

Existing solutions differ on the techniques they use to encode each part of the timestamp and on the size (number of parts) of the multipart timestamp. Furthermore, they also differ on the entity that is in charge of assigning timestamps to each update.

2.3.1 Timestamp Part Encoding

In a multipart timestamp, all entries have the same encoding. According to the state of the art, there are three different approaches: *logical clock*, *physical clock* and *hybrid clock*.

Logical Clock. A logical clock is comparable to a counter and is incremented after each update. It is used on COPS [2], Orbe [3], PRACTI [4], Lazy Replication [5], ChainReaction [6], SwiftCloud [7], Occult [14] and C^3 [16].

Physical Clock. A physical clock is like a watch, because it is monotonically increasing. Usually, it is the time since some important date (on Unix, it is counted from 1 January 1970) and is used on Legion [8], Cure [9], GentleRain [12], POCC [13] and Saturn [15].

Hybrid Clock. A hybrid clock combines the previous two. When the physical clock part is delayed (a machine has a faster clock than the other), the new value is obtained by incrementing the logical clock. This is a way of avoiding unnecessary waiting time to reach a consistent value. It is used on Okapi [10] and Eunomia [11].

2.3.2 Timestamp Size

As previously mentioned, different systems adopt timestamps with a different number of parts. The smaller they are, the smaller the network bandwidth needed for exchanging them. Moreover, the storage

space occupied with metadata is also smaller. However, it introduces a penalty in the visibility latency of concurrent operations. In the following sections, this issue is explained and it will be discussed with the edge computing paradigm in mind.

In the survey of the existing solutions, there are four different sizes: *one part per object*, *one part per server*, *one part per datacenter* and *one part for the entire system*. In the remainder of this section, the analysis is supported by an example.

2.3.2.A One Part per Object

Figure 2.1 has an example of this strategy. Different objects have independent logical clocks, but their dependencies must be related to prevent an incorrect update order.

The client's causal history is updated after reading or writing objects. If the read object has a newer version than the one previously seen, then the client updates the corresponding part on its causal history timestamp. Otherwise, it stays the same. After writes, the client can clean its causal history, by only storing the creation timestamp of that update. Taking the causal consistency transitivity rule into account, this compression is considered safe, because all future operations will potentially depend on the current update, with all of its dependencies. These dependencies are obtained by the client's causal history at the time of the operation.

The update creation timestamp is calculated in the receiving server, after it has equal or newer versions than the ones in the client dependencies. It gets the highest timestamp for any local version of that specific object and increments it by one. The version identifier is stored with the new object value.

Returning to Figure 2.1, Op_3 depends on Op_1 and Op_2. This is expressed by keeping version 1 for object A and 4 for object B in Client$_1$'s causal history. Because Server$_1$ stores the version 1 for object A, the update timestamp will be 2. Afterwards, Client$_1$'s causal history only stores the version 2 for A.

Updates that come from other replicas (*remote updates*) are processed in a similar fashion. Firstly, they must wait for, at least, the object versions in their dependencies to be present. Then, they are made visible, but remain with the version that was assigned by the original replica. On the example, both servers can immediately apply the remote updates, because they store the object versions in the updates' dependencies.

One system that adopts a similar kind of dependency tracking is COPS [2].

2.3.2.B One Part per Server

In Figure 2.3, both servers have a full replica of all objects in the system. For simplicity, there are only two objects: A and B.

Each multipart timestamp keeps two entries for identifying the versions. The first belongs to Server$_1$ and the second to Server$_2$. Clients track their causal history by storing the highest partwise timestamp

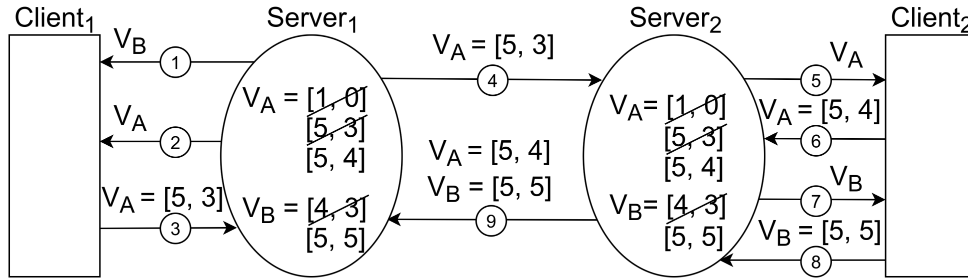

Figure 2.3: Causal consistency with timestamps with one part per server.

ever seen. It can come from reads or after writes. When a server receives an update from a client, it creates the new timestamp by incrementing its clock and replacing its part with the new clock on the client dependencies timestamp. This multipart timestamp is then used for tagging the update and updating the client dependencies.

For instance, after Op_1, Client_1's causal history is tagged with the creation timestamp of the available version of object B ([$4, 3$]). Op_2 does not affect the dependencies, because the returned version of A has partwise smaller values than the version of B. Op_3 updates the version of A on Server_1. The associated dependencies are tagged with the client dependencies timestamp. The new version timestamp keeps Server_2's part, but changes Server_1's entry. Given that the current clock is at 4, its entry will be set to 5, resulting in [$5, 3$].

After receiving the remote update from Server_1, Server_2 can immediately make it visible, because it has received and applied all the updates until the highest predecessor of the Server_1's timestamp part. Otherwise, it would have to wait for this condition to be fulfilled.

A system that follows this strategy is Lazy Replication [5]. PRACTI [4] and Legion [8] are similar, but allow partial replication of the objects. Orbe [3] has an alternative representation, displaying the different parts as a matrix. It assumes full replicas over datacenters and each datacenter has equal partitions to distribute the objects. So, the matrix has N rows (one per partition) and M columns (one per replica).

2.3.2.C One Part per Datacenter

Following Orbe's [3] perspective, the size of the timestamp can be reduce even more. If there is a compression of all the operations of all servers inside a datacenter to a single part of the timestamp, it will have as many parts as the number of datacenters.

As in the previous category, the client's causal history timestamp is updated during both operations, storing the partwise maximum of the returned timestamps. However, the update creation timestamp may not come from incrementing the local server clock. In spite of being one possibility, there must be the guarantee that values are monotonically increasing. So, the server may replace its current clock if its datacenter part in the client's causal history has a higher value. Then, it increments it by one, swapping

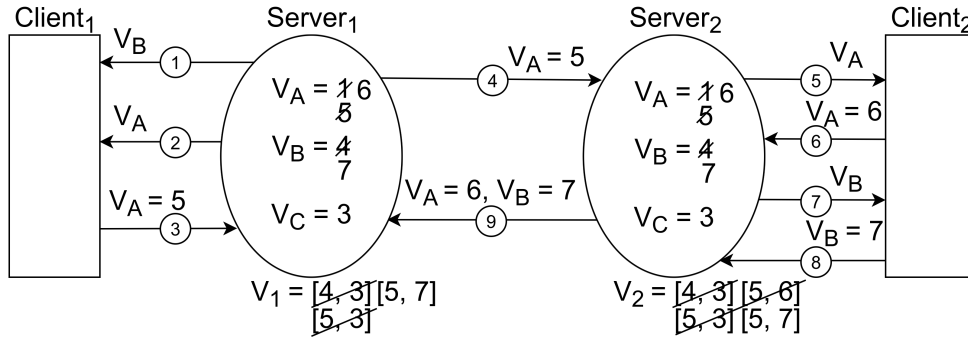

Figure 2.4: Causal consistency with timestamps with one part for the entire system.

the corresponding part.

Considering that both Figure 2.3 servers belong to the same partition on different datacenters, the displayed timestamps would be the same. In case of a client contacting a different partition, the result could be different, because it could have a newer part value for the current datacenter. When performing a write, the receiving server would eventually advance its local clock value.

Remote updates follow a similar approach as in the previous category. Usually, there is either an intra-datacenter protocol for calculating the latest updates received from foreign datacenters or First In, First Out (FIFO) channels for communication. Both of these strategies ensure that there are no older missing updates.

Examples of systems that use timestamps with this size are: ChainReaction [6], SwiftCloud [7], Cure [9], Okapi [10], Eunomia [11], POCC [13] and C^3 [16].

A small variation to this technique is to have one part per partition. One implication of such choice is the change of the internal structure of the datacenter, which will be explained in the following sections. The system that proposes this alternative is Occult [14].

2.3.2.D One Part for the Entire System

The different versions of an object are identified by a timestamp with only one part. It has a fixed size, because it does not depend on the number of objects, servers, replicas nor partitions.

The client's causal history is represented by another timestamp. It is the maximum timestamp ever seen by this client. On reads, it is only updated if the returned object version has a higher timestamp. For writes, it is always changed, because the returned server timestamp is guaranteed to have a higher value.

Following the example in Figure 2.4, Client$_1$'s causal history is represented by 4 after Op_1. Op_2 does not affect it, given that the returned timestamp is smaller. The output timestamp of Op_3 must be larger than the one of the Client$_1$'s causal history and unique in Server$_1$. So, Server$_1$ assigns the timestamp 5 to the operation.

Remote updates are only safe to apply after being sure that all datacenters will apply them. To track it, servers still need to manage a vector-like structure with one entry per datacenter. Each entry stores the latest update timestamp received from the corresponding datacenter. Over time, the intra-datacenter servers exchange their vectors and compute the global minimum. The calculated value can be seen as a marker, allowing the application of remote updates with equal or smaller timestamps. In order to simplify the example, figure 2.4 does not show this situation.

One example of this kind of dependency tracking is GentleRain [12].

Saturn [15] introduces a novel design, considering that each datacenter has a specialized component for creating the timestamps. In the system, the timestamps are called *labels* and each server update is tagged with one. Local updates are immediately visible, but remote updates have to wait until the corresponding label and data are received. Causal consistency is ensured by a predefined topology for propagating the labels between different datacenters.

2.3.3 Timestamp Assignment

In distributed systems, the simplest approach for creating a solution is to *centralize* it. For the assignment of timestamps, it is one possibility. However, there are two more: *centralized server in the datacenter* and *server contacted by the client*.

Centralized Service. Kronos [1] is a centralized service that is able to manage causal relations between any abstractions. Clients are responsible for creating events and giving them an order. The service can be contacted to get a consistent sequence of operations.

Centralized Server in the Datacenter. The servers that store the objects contact a different entity inside the datacenter in order to tag updates. By doing this, there is no need for an intra-datacenter synchronization protocol. Saturn [15] is a system that uses such strategy.

Server Contacted by the Client. On the majority of the surveyed solutions [2–14, 16], the server contacted by the client is able to directly timestamp the update. Some of them need *stabilization protocols* before applying remote updates.

2.4 Causally Consistent Systems

In this section, there is a description of each surveyed system. It starts by explaining how a client reads objects. Then, it goes through the client writes and the chosen strategy for exchanging the updates among the replicas. Finally, there is a comparison of all these systems, in order to identify key characteristics that might be interesting for the edge.

2.4.1 Kronos [1]

Kronos is a centralized service that keeps a *dependency graph*. Nodes represent *events* and edges define the causal relations between events.

When a client wants to obtain a *reference* to an event, it must contact Kronos. A reference is an identifier of an event and it is managed by the service. After gathering a set of references, clients must know a possible order for applying the associated events. So, they must contact Kronos, by providing the set as an input. The output is a sequence with all the items in the set that does not violate causality. Different clients may get different results, because concurrent operations can be in a different order.

The addition of events to the graph also takes two phases. In the first, a client creates the event, receiving its reference. Then, it is able to assign an order, using the new event reference together with the other existing ones. There are two ways of ordering the events: *must* and *prefer*. Must specifies a strict ordering: either all must conditions inside a call are fulfilled or the operation is aborted. The prefer ordering is used for assigning relations between concurrent operations. Its validity is only checked after Kronos analyzes the must conditions. If they introduce inconsistencies, the operation does not abort, but the ordering is discarded.

Given that this is not a geo-replicated solution, there is no remote servers to send new updates. This contributes for a simpler design, but creates a performance bottleneck.

Kronos is an isolated system that only deals with the causality of events. This introduces a novel abstraction, because different events may have very different granularities.

The biggest weakness is related with the way clients get a correct ordering of events. If a client has not received a reference that depends on other that was sent to the server, then the ordering will create an inconsistent client state. So, either the client must have all the references to get a global ordering or there must be an application dependent protocol to prevent this situation.

In spite of having an interesting approach to ensure causal consistency, Kronos cannot be deployed on the edge. It is a centralized service that would quickly become a bottleneck, violating the scalability requirement.

2.4.2 COPS [2]

COPS has a timestamp with one part per object. The operations follow the same steps as described in Section 2.3. Each part has a logical clock.

Clients are in charge of managing their causal history. In COPS, this history is called a *context*. Every time a client reads a new object, it adds the version to the context. Datacenter servers always give the latest consistent object version, without looking at the client context.

When a client wants to write an object, it starts by using the context to calculate the *nearest depen-*

dencies. The context can be seen as a dependency graph and the transitivity rule is used for finding these dependencies. A nearest dependency is obtained when a version of an object does not happen before any other version. Afterwards, the client sends the object's key, new value and the nearest dependencies to the local server. This server has all the dependencies, because the client keeps a session with it. So, it can immediately apply the update, tag it with the dependencies and with the new timestamp. This timestamp is returned to the client and it replaces all the previous context.

The remote update protocol follows similar steps to the one in the local client update. When a server receives an object from other replica, it must check if the dependencies are fulfilled. Unlike the previous situation, there may exist other updates that belong to the dependencies and are not yet present. So, the receiver must delay the update until it can be safely applied.

An interesting feature is that a client can have different contexts for the same server connection. Given that dependencies are tracked per client context, it allows independent operations to have a lower amount of common dependencies, reducing the time remote updates have to wait before being visible (*visibility latency*).

The timestamp format introduces some compression of metadata, visible by existing only a scalar per object and not a vector with an entry per replica. This results in a similar amount of metadata as in Kronos, but with a higher visibility latency.

COPS was one of the first systems to provide causal consistency over the cloud storage. However, the metadata format is not prepared for the size of the network edge. Usually, application workloads tend to include a much higher number of reads than writes. In this solution, the processing power needed for checking the dependencies of remote updates could bottleneck the nodes. Furthermore, it does not consider partial replication, as it was not a concern at the time.

2.4.3 Orbe [3]

This system explores the datacenter layout for choosing the number of timestamp parts. Each datacenter is divided in N partitions and there are M different replicas. Considering that each partition is assigned to a different server, it is possible to address it through a matrix-like structure with N rows and M columns. Clients track their causal history with a copy of this matrix. It is filled with the logical clock of the corresponding server.

A client read only sends the key of the object. The server replies with the latest available version's value, creation clock and the replica index. If the entry pointed by the index has a smaller value than the returned clock, then the client overwrites that position with the newer value.

To write an object, a client sends the key, the new value and its matrix to the local server responsible for the partition. The matrix corresponds to the update dependencies and is used on replication. Before tagging the creation time, the server has to increment its local clock, guaranteeing that the assigned

value is unique. After storing both the object's value and metadata, the server replies with the update's clock and its index. Similarly to COPS, the client resets its causal history to default values, only saving the returned clock at the given index.

For replicating updates, each partition talks with its peers from different datacenters. The updates are sent by their creation order with all the associated information (key, value, creation clock, dependency matrix and replica index). To track the applied remote updates, there is a vector with one entry per replica and it has the clocks from the latest updates from each replica. The receiving server uses the matrix and the vector to verify the dependencies. For its partition, it looks at the corresponding row of the same replicas and compares it to the vector. If the vector is larger than or equal to the row, the server can go to the next step. For other partitions, every time there is a value different from the default one in the partition column, it means that the operation depends on other elements that the current datacenter may not store. So, the server contacts the servers that are in this situation, by making an *explicit* dependency check. If they fulfill the dependencies, the remote update can be made visible and the corresponding vector's entry updated.

As an attempt to reduce the amount of metadata, the authors only store the matrix entries that are different from the default ones. This strategy allows clients to only store the contacted servers' clocks as a dependency for future operations. However, once a client reads a newer version from a server, the matrix must be updated in the corresponding entry. This can lead to a matrix that has all the entries filled, resulting in no gain for the worst case.

Given the coarser grain in which the dependencies are tracked, the impact of false dependencies will be greater. For instance, updates on one server will depend on all the previous updates applied in the same server. So, even when there is not a direct version dependency between objects, it will be treated as so. The increased waiting time before being able to apply the updates will have an instant impact in a higher visibility latency.

Although Orbe is able to bound the metadata size, it is not independent of the amount of participants. Like COPS, Orbe requires replicas with the full system state. Both characteristics would not work on the network edge.

2.4.4 PRACTI [4]

The name of this system includes its main features: partial replication (*PR*), arbitrary consistency (*AC*) and topology independence (*TI*). The granularity of partial replication is the finest, because each replica chooses the individual objects it wants to store. Like Orbe, the timestamp has one part per server and it uses logical clocks.

When a node wants to participate in the network, it chooses three sets of other nodes: one for receiving the updates, other for prefetching objects for its *interest set* and another for requesting newer

consistent versions of the locally stored objects. The interest set is any group of objects the node stores and it may have one or more interest sets.

Each node stores a global timestamp and as many timestamps as interest sets. They are updated at different times, expressing different things. They will be analyzed when talking about the replication of updates.

Nodes read their local copy of the object when it is *valid*. The validity is checked on the timestamps and will be explained on the replication of updates.

Like reads, writes are done to the local replica. Each version is tagged with the value of the node's logical clock. Afterwards, it is sent to the nodes that subscribed to the object updates.

The replication of updates is done by two types of messages: *invalidations* and *bodies*. The first kind contains only metadata and can be either *precise* (one message for object) or *imprecise* (one message for a group of objects and/or multiple updates), but must follow a causal ordering of delivery. Imprecise messages have a *target set*, *start* and *end* clocks. Bodies have precise metadata and the actual data, and they do not need a specific order of delivery.

Upon receiving invalidation messages, the node may update the object's interest set timestamp as well as the one from the local node. The first is only updated with precise invalidations, but the second joins both. When the second has a newer version, the local state is *imprecise*. If the body for a precise invalidation has not yet arrived, the interest set's state is *invalid*. Invalid objects cannot be read, because it prevents inconsistencies for stronger models. However, for causal consistency, the older values can be returned, meaning that there are stale reads.

Supporting different levels of consistency has an impact in the amount of metadata and on the time taken to apply each update. Given that all the other systems are only defined for causal consistency, this will not be referred as a negative characteristic.

Although nodes can choose to keep only a subset of all existing objects, they must receive all invalidation messages. This is needed, because of the unstructured topology, which could otherwise lead to inconsistent information on the nodes. To reduce the impact of the communication overhead, imprecise invalidations work as a summary of updates whose data is not locally stored. Still, nodes receive information about objects that they do not replicate, which creates a *non-genuine* partial replication. If all messages were about replicated objects, then it would be *genuine* partial replication.

PRACTI is the first system to unveil some promising strategies that could be used for edge computing. The topology independence is particularly interesting, because new nodes do not have to be integrated in a rigid structure by means of a joining protocol. Moreover, there is support for partial replication, allowing machines with less storage to participate. However, metadata grows with the number of replicas, which could become unbearable at the edge.

2.4.5 Lazy Replication [5]

Each node has full replication of the data, meaning that it must store all the objects in the system. Like PRACTI, there is not a predefined network topology and the timestamps have one part per node.

In Lazy Replication, there are two types of timestamps: *Unique Identifiers (UIDs)* and *labels*. The first tags updates and the second compresses multiple UIDs. Labels create the client causal history and the dependencies for all operations. Servers also keep timestamps for tracking their current version.

When reading an object value, the client sends the key and its label. The server waits until its state includes all the client history. The response has the latest object value and the corresponding timestamp. Before delivering the value, the client updates its label by including the partwise maximum between its previous label and the received timestamp.

When a client tries to write an object, it is constantly contacting the known entities by sending the object key, new value and its label. Possibly, it may send it to different ones and create duplicates. Given the desire for *at-most-once* message delivery, each client has its Client Identifier (CID). It is included in the other update metadata, allowing different replicas to recognize the duplicate updates. On the server side, if it is a new update, it increments its clock, updating its local label. The update UID is obtained by replacing the local server's part of the client label by the new clock value. As a response, the server sends the update UID, that is able to replace the client label while keeping the causal history. Finally, the client acknowledges the reception, marking the completion of the operation.

Updates are replicated through *gossiping*, ensuring that eventually every node sees them. The adopted strategy is a *log replication*. Each replica has its own log with all the operations that changed the local state. After some period of time, the replica sends it to its direct neighbors, which will filter duplicate entries and apply the missing operations. Filtering is done by looking at the update UID with the CID.

As a garbage collection strategy, messages that take more than a well defined time interval to arrive are discarded. The acknowledgments of the client updates are included in the log, because they work as a marker to prune the older log entries. If future acknowledgments are discarded by the time condition, the size of the log is reduced.

Lazy Replication is not ready for the edge, for similar reasons as the previous systems. However, it introduces the use of unique identifiers that filter duplicate updates. This is interesting in the perspective of fault tolerance. Given that the network edge is much more dynamic, the probability of clients needing to send the same request more than once is not negligible and, with the presented strategy, the duplications would not create inconsistencies.

2.4.6 ChainReaction [6]

As the name hints, ChainReaction is built over a chain replication architecture. In the following paragraphs, there is an explanation on how it affects the different operations. This system uses timestamps that have one part per datacenter. Each part has a logical clock that belongs to a server of the corresponding datacenter.

Clients track their causal history through a table with one entry per viewed object. The version timestamp and the *chain index vector* are the metadata associated with the object's key. The chain index vector has as many entries as the number of datacenters and it stores the latest position in the chain that the object has been replicated to.

If the chain index entry for the local datacenter is equal to the predefined chain length, the client may send a read request to any node of the chain. All of them will be able to answer the request without waiting for a remote update. Otherwise, the request can be made to any server until the one that is specified in the entry. When the client changes datacenter, the previous object version may not be available. So, the head of the corresponding chain must wait for it or make an explicit request to a datacenter that has it. The server returns the value with the version timestamp and the local datacenter chain index. If it is a new version, both entities are changed. If not, the chain index entry for the local datacenter keeps the highest value.

Every write is sent to the head of the chain and propagated to the following nodes. In the message, the client sends the object's key, new value and a compression of all accessed objects' metadata since the last update. The head of the chain starts by assigning a new object version by incrementing its replica part of the timestamp. Afterwards, the update is sent to the other nodes until it is *k-stable*. A *k*-stable update is replicated in, at least, *k* nodes of the chain. Only then, the head is able to return to the client the object version and the index of the latest receiving node.

When an update reaches the tail of the chain, it is *DC-Write-Stable* for that replica. Updates are delayed until every object that they depend on is in this condition, preventing clients from reading inconsistent versions. DC-Write-Stable objects do not need to be kept in the client's accessed objects table. If all chain replicas are in this condition, the update is *globally* stable.

Remote updates are scheduled immediately after a chain head receives a client update. For this replication, there is only needed the update timestamp. The datacenter has a specialized entity responsible for exchanging these updates and it is called a *remote-proxy*. It also has a timestamp in order to track the previously applied updates. When a remote update arrives, its timestamp is compared with the one of the receiver's proxy. If the latter's timestamp has at least the same entry values for other datacenters and one less for the sending datacenter, the corresponding dependencies are stable on the current datacenter. So, this update can be applied. Otherwise, it must wait until these conditions are verified.

It is important to note that a node may have different positions for different objects: for one it may be the head, for other the tail and for another be in the middle. Different types of nodes need different fault tolerance strategies. The head is substituted by the second node, the tail is not replaced and the middle nodes are hopped over. The previously chosen *k* for a *k*-stable update works as the minimum number of nodes for ChainReaction to work properly.

This system relies on a very strict topology for replicating the updates, which could raise performance issues at the edge. Moreover, the tail of the chain receives remote updates much later than the other nodes, because there could be many hops before it is reached. As in previous systems, the metadata size is still an issue.

2.4.7 SwiftCloud [7]

The timestamp has one part per datacenter and one more for the client. The client part is necessary, because each client keeps a local object cache and may send the updates for different datacenters, before they are acknowledged. This is a similar strategy to the Lazy Replication's CID. Each timestamp part stores the corresponding entity logical clock.

When a client wants to access SwiftCloud, it must maintain a session with the firstly contacted node. Clients request objects they want to cache and include them in their *interest set*. All operations are done in the cached objects and may update the client's causal history timestamp.

A read gets the locally available value and merges the update timestamp with the client's history. Updates are tagged with dependencies (a copy of the client's timestamp) and with an initial timestamp of the client's clock. In the background, the new updates are sent to the datacenter node by their chronological order. After receiving it, the server reassigns a timestamp with its logical clock and makes the update visible to other clients.

For delivering remote updates, the node has the client's current version vector and only sends it the changes that the client has not seen. If a client wishes to change the number of elements in its set, it must notify the associated node. When there are new objects to replicate, the node delivers the most recent consistent versions like remote updates. In both situations, the client's causal history timestamp is updated.

Regarding inter-datacenter replication, the receiving datacenter must wait until the local version is consistent with the update dependencies. When these updates become *k*-stable, they can be sent to the clients that want to replicate them. To detect which updates are *k*-stable, each datacenter stores a version vector that is causally updated when it receives $k-1$ acknowledgments from other datacenters.

To reduce the size of the timestamp, the dependencies may not take into account all the existing datacenters. They may only mention the ones that were read/written before the update.

SwiftCloud's *k*-stability may be a promising fault tolerance measure for an edge deployment. It has

a negative impact in the remote update visibility, but avoids causal consistency violations when replicas fail. The delivery of remote updates to the client's cache is not best option for the edge, given that replicas have to keep specific state for each client. This approach impairs scalability. Also, clients are the only partial replicas, meaning that servers must store all the objects.

2.4.8 Legion [8]

The main motivation of this system is to change the interactions in collaborative applications. In the past, there were datacenters with replicas of the objects and the clients had to use them as a middle-man. This results in high delays to receive the updates and blocks the dissemination of changes when the datacenter is unavailable. As a result, Legion allows the direct communication between clients.

A client device does not have the available storage to be a full replica of a datacenter. To circumvent this problem, each client replicates a subset of all objects and organizes them in one or more *containers*. For instance, a container can have all the objects that belong to some collaborative work. Although clients may use a subset of the container, they still have to replicate all of it. This can be seen as a non-genuine partial replication method, like the one from PRACTI. Each container object has its own timestamp with one part per replicator node. Unlike the previously presented systems, Legion has entries with physical clocks.

When a client wants to join a container, it must connect to a number of *near* and *distant* nodes. They are chosen by their latency to well known servers, which is calculated by the RTT of ping messages. The difference between the two kinds of nodes allows a lower visibility latency for updates as well as a higher tolerance to network partitions.

Reads and writes are applied to the locally available object versions. A write creates a new object version, which timestamp is changed on the local node entry to the current clock value. At the same time, the update is added to the container's *version chain*. This structure enables the transmission of only the differences among the old and new container states, by encoding them with CRDTs and transferring them in the background. Between each client pair, there is a FIFO channel, which preserves the causal ordering on the replication of updates.

In order to cope with legacy applications, there is a special client that is in charge of the communication with the datacenter. Older applications may not have the changes for the direct client communication, leading to a client sending local updates only to the datacenter. The special node maintains the global consistency by sending the changes from the inter-client communication to the datacenter and disseminating newer datacenter versions among its peers.

The possibility of clients propagating the updates among themselves is an attractive prospect for the edge of the network. It could reduce the load on the servers and fasten the dissemination of updates. However, the synchronization between the nodes might not allow the current version of the protocol to be

used at the edge. Furthermore, the multipart timestamp has one entry per replicator node. This means that more popular objects have more metadata, which may not be possible to handle in an efficient way.

2.4.9 Cure [9]

In spite of not being one of the goals of this thesis, this was the first system to successfully allow causally consistent transactions with both object reads and writes. Other solutions allowed transactions with either one or the other.

The causal dependencies are expressed by a timestamp with one part per datacenter and physical clocks. However, the timestamp creation process is more complex than the ones of previous systems. It is explained when presenting the transactions that include object updates.

Before a client can access objects, it must contact a *coordinator* to obtain a transaction identifier. A coordinator is any server of the local datacenter that is responsible for committing the client transaction. The identifier works as a boundary for the updates that a client can access. During a transaction, the client can see objects whose version has a lower timestamp than the one from the identifier.

Clients can read one or more objects at the same time. However, they keep the transaction identifier as immutable metadata. Regarding writes, a client may also make more than one at the same time, but they are not final. They will only be stored after a successful commit.

When a client commits a transaction that has object writes, all of them will have the same timestamp, so that the atomicity property is guaranteed. The timestamp calculation involves the partitions whose objects were updated. Each partition proposes the current value of its clock to the coordinator. After gathering all the proposals, the coordinator chooses the highest value and notifies the participating partitions of the result. It also replaces the content of the local datacenter entry in the dependencies to create the update timestamp.

The replication of updates is a pairwise process between the same partitions on different datacenters. After a specific time interval, pending committed updates are transmitted to other datacenters. Replicas have two timestamps for controlling the visibility of this kind of updates. The first tracks all the updates (both local and remote) and the second is used for showing the remote updates. The receiving server advances the sender's timestamp entry to the update timestamp on the all operations structure. To know which remote updates can be made visible, partitions in the same datacenter periodically exchange their all operations timestamps to compute the second timestamp. Each part will be the minimum value of the received timestamps for that specific entry. Remote updates whose dependencies have lower or equal timestamp than the global timestamp can be made visible.

The most obvious drawbacks to the adoption of Cure on the network edge are need for full replication and the metadata size. When looking a little bit further, the transactional scheme is another issue. It requires a fair amount of synchronization among the nodes, that, in the edge of the network, may lead

to a non-negligible number of retries. So, it is preferred to use simpler operations on this setting.

2.4.10 Okapi [10]

Clients track their causal history using timestamps with one part per datacenter. Servers have three timestamps of the same size: one for all the received updates, other for globally stable updates (similar to Cure) and another for *universally* stable updates. An universal stable update is an update that is globally stable in all datacenters and its importance is explained when addressing the replication of updates. Okapi uses hybrid clocks as entry values.

In total, clients have two timestamps and a *dependency time*. One is a consistent version of a local server's universal stable timestamp and the other is a merge of the previous one with the dependency time in the local datacenter's entry. A dependency time is returned for objects that are not yet universally stable, but were locally created. It is the clock value of the server in which the update was created.

When a client wants to read an object, it sends the key and the local copy of the universal stable timestamp. If the server is behind, it assumes the received timestamp for its local copy. At maximum, the returned version has a timestamp as large as the client's timestamp. With the object value, the server also returns its universal stable timestamp and, as said in the previous paragraph, possibly the dependency time, which are stored at the client side.

Writes are appended with the new object value and the merged client timestamp. The update creation timestamp is assigned at the server, by using the latest clock value in its corresponding entry of the merged client timestamp. The server timestamp for all updates is also updated in its entry. As a confirmation, the client receives the new dependency time associated with the update.

The replication of updates is similar to the one in Cure. However, the primary replica only sends its part of the timestamp. The receiver knows which entry to update, because there is a pairwise communication. The global timestamp is calculated as in Cure. The universal timestamp uses the global ones, which are now exchanged between the same partitions of different datacenters. As previously, each entry is the minimum value of all the exchanged timestamps. The universal timestamp guarantees that clients observe the same updates when changing datacenters, unlike what happens in Cure.

When considering the transition to the edge, Okapi struggles with full replication and the metadata size. Moreover, the protocol for the propagation of updates is too heavy, requiring a lot of stabilization among the nodes. Edge nodes do not provide any availability guarantees, which could result in their disappearance blocking the application of updates.

2.4.11 Eunomia [11]

At the time of the proposal of this system, the two most common types of mechanisms to enforce causal consistency were: *sequencers* and *global stabilization procedures*. The first had the problem of being in the client's critical path, introducing a delay for the completion of the operations. The second uses more complex dependency checking, which needs more communication between partitions of different datacenters.

In order to combine the best of both approaches, Eunomia uses a *site stabilization procedure*. There is a new component in the datacenter, which is responsible for exchanging remote updates with other datacenters. When the current datacenter has the primary replica, the update is transmitted to this new component in the background. The process is detailed in the propagation of remote updates.

Each client keeps a timestamp with one part per datacenter, storing the highest hybrid clock for each datacenter. The Eunomia service has a timestamp for controlling the latest applied remote updates and a vector with one entry per partition.

When a client wants to read an object, it sends the key to the partition server. The server returns the object's value and creation timestamp. The client keeps the partwise maximum between the received timestamp and its own local timestamp.

For a write, the client sends the object key, new value and its local timestamp. This timestamp is used as the update timestamp basis. It is only changed in the local datacenter entry by giving it a higher value. The server forwards both data and metadata to the Eunomia service and the update timestamp to the client. The client can replace its local timestamp, because the new one is guaranteed to be larger.

The replication of updates only happens when Eunomia is sure that it will not receive any update with a lower timestamp from a local partition. It keeps track of the datacenter clocks of previously received updates on the partition vector. Every update that has a lower value on the local datacenter entry than the minimum value of all vector entries is safe to be replicated to other datacenters. For a datacenter that receives a replicated object, it puts it in the sender's pending updates queue. After a certain period of time, there is a dependency check to make those updates visible. If, for every entry that does not belong to neither the sender nor the receiver, the local timestamp has a higher or equal value than the update timestamp, then the update is applied. The local timestamp is accordingly changed to allow future updates. It is important to note that queues are sorted with the earlier updates on the head and the newer on the tail.

Although Eunomia does not support partial replication and its metadata is not constant, the propagation of updates uses a lightweight protocol. Nodes have all the information for independently choosing when to make the updates visible. This autonomy is valuable for edge devices, mainly due to increasing the resilience to the failure of foreign nodes.

2.4.12 GentleRain [12]

Unlike the previous systems, GentleRain tags updates with a timestamp with one part for the entire system. This means that it always has the same size, independently of the number of objects or participants. Regarding the clock type, it uses a physical one.

Clients store two timestamps: a *global stable time* and a *dependency time*. The first is calculated by the servers and acts as a marker for making remote updates visible. The second is used in some interactions with the local datacenter. Servers maintain a vector with one entry per datacenter that allows the computation of the global stable time. There is also a *local stable time* that works as an intermediate result. Their differences are made clear in the replication of updates.

When a client wants to read an object, the request includes the key and the client's global stable time. If the server's value is smaller than the one provided by the client, it replaces its local copy of the global stable time. The response has the newest object version with a timestamp lower than the global stable time, its value and the server's global stable time. The object's update timestamp is stored in the client dependency time if it is larger.

For writes, the client attaches its dependency time to the object's key and new value. The server responsible for the object must guarantee that the update has a larger timestamp than the client dependency time. This may result in blocking the operation until the internal server clock respects the condition. Afterwards, the clock replaces the current datacenter entry in the server's vector and is assigned as the update timestamp. The server returns its update clock to the client, which will replace the dependency time. At the same time, the server enables the background replication of the update.

Each remote datacenter entry on the local version clock is updated when an update coming from that specific datacenter arrives. It will be changed to the update's timestamp value. However, there is a possibility of not making the update visible, because of the lower value of the global timestamp. The calculation of the global time has two intra-datacenter stages. The first is to disseminate on every time interval the minimum entry value of each partition's local version vector (local stable time). After gathering all values, each partition can get the global stable time, which is also the minimum of all the exchanged values.

GentleRain is the first surveyed system that tags updates with metadata of constant and small size. However, it still does not offer partial replication and, internally, the stabilization protocols rely on information with one entry per datacenter. The improvement over the update metadata size is overshadowed by the structure used on the global stabilization protocol, disallowing its transition to the edge.

2.4.13 POCC [13]

This system has an optimistic approach on causal consistency. Every time servers receive an update (client or remote), they directly include it in the version chain. To ensure causal consistency, clients are in charge of managing the returned value to the application in their upper layer. The authors argue that there is no need for servers to keep heavy dependency check mechanisms, because clients *usually* ask for updates that are already replicated.

Clients have two timestamps with one part per datacenter. One tracks the dependencies of all operations and the other is updated on reads. Servers have a timestamp that controls the stored objects. Each entry stores a physical clock value.

For joining the system, a client creates a session to a particular server in the local datacenter. Either the server has the objects the client wants to access or it forwards the operations to a responsible one.

When a client tries to read an object, it sends the object's key and its read timestamp. If the server's timestamp is larger or equal to the client's read timestamp, it can answer the request with a consistent object version. Otherwise, it must wait until it receives a new version from other replica. The server replies with the object's value and all the associated metadata. The most important ones are the object dependencies (it will update both the read and all operations timestamps of the client) and creation clock (it only updates the all operations timestamp).

It is possible to identify a situation in which a read never terminates. For instance, if a client depends on an update from a remote datacenter and there is a network partition that separates both datacenters, then the client hangs for an indeterminate amount of time. To solve this problem, the client has a time interval to complete the operation. Otherwise, it starts a new session to the current datacenter, but in a pessimistic approach (similar to the one in Cure).

To write to an object, the client sends the object's key, new value and the all operations timestamp. The server waits until its current clock is higher than the maximum value of the received timestamp, ensuring a higher clock than any of its dependencies. The clock value will replace the local datacenter entry in the server's timestamp and tag the object creation clock. The object dependencies are expressed by the all operations client timestamp. The server responds to the client with the update creation clock, which replaces the local datacenter entry in the all operations timestamp.

Updates are asynchronously replicated by their creation clock order. The receiver server adds the object version to the version chain and updates the local timestamp on the sender's entry to the update creation clock.

In spite of not fulfilling the metadata size and partial replication requirements, POCC presents a shift in the strategy for enforcing causal consistency. The optimistic approach may be reasonable for a causally consistent edge store, because clients can retry in other replicas and, sometimes, continue to operate in the presence of failures to their local replica. However, the transition to the pessimistic

solution raises some previously discussed inconveniences.

2.4.14 Occult [14]

This system follows an optimistic approach. The timestamps have one part per partition, instead of per datacenter. It is the only system to adopt this structure, which leads to a master-slave replication technique. Each entry stores the logical clock of the corresponding partition. Clients have one timestamp for tracking their causal history. Servers only keep their logical clock. However, updates are stored with their dependency timestamp.

In order to distribute the load of read operations, a client can contact any partition replica by sending the object's key. The contacted server immediately answers with the latest object value, its dependency timestamp and the current logical clock. Afterwards, the client must check if the version is consistent with its causal history by comparing the logical clock with the corresponding partition entry value. The inconsistency may occur when the update was made on the master and it has not yet reached the current replica. So, a client can try more times at the same replica and, if it is still unsuccessful, try the master instead. The last attempt is always successful, because the master has the most updated version of the partition data. Finally, the client updates its own timestamp to reflect the new dependencies.

Writes are always sent to the master server of the partition. The request includes the object's key, new value and the client timestamp. The master uses the client timestamp to derive the update timestamp, by assigning the latest logical clock to the corresponding partition entry. This clock value is also returned to the client, which will be used to update its timestamp.

Updates start being replicated before a master server returns from a client object update. They are sent to the slaves with the associated timestamp and the master clock. The second will be set as the slave's current clock value. All updates are replicated by their creation order.

Given that the number of partitions tends to be very large, there are three optimizations to the timestamp size: *structural compression*, *temporal compression* and *isolating datacenters*. As a tradeoff, the compression of metadata leads to a higher amount of false dependencies, affecting the visibility latency of updates in a negative way.

The structural compression works as a hash function. The timestamp is previously defined and it should be much smaller than the number of partitions. Afterwards, each partition occupies the position given by its number modulo the size of the timestamp. In spite of different partitions being mapped to the same entry, it only stores the maximum clock value. This avoids a client from reading an inconsistent version of an object, but may result in a client blockage. If a partition has less updates than other that is also mapped in the same entry, the reads will always fail, because the client depends on a *newer* version. So, the clocks of all partitions must be synchronized.

The temporal compression also uses a predefined number of entries, but keeps detailed information

for the most recently used partitions. If a timestamp has N entries, then the first $N-1$ entries will store the partitions with higher clocks and the remaining entry will compress all the other partitions by storing their maximum value. This is a dynamic structure that can be updated on every read and update. On reads, the client must have two sorted vectors by decreasing clock value and choose the maximum entry from both, creating a new vector with a possibly third different order. On writes, it is easier, because there is only the returned partition clock to compare with the vector entries. Starting from the last position, when there is an entry with a larger value than the received one, then it should be on the previous position. The compression of the values of the remainder partitions must also be considered.

The final optimization tries to reduce the visibility latency of updates when the slave and the master are on different datacenters. Each datacenter has its own timestamp, which, in the worst case, can be seen as an approximation of Orbe. On reads, all client timestamps are updated by looking at the timestamps returned by the server and getting the pairwise highest entries from the timestamps of the same datacenter (one that the client already had and the other sent by the server). On writes, clients only change the timestamp of the datacenter that hosts the master shard.

Although the authors consider a fully replicated data store in their evaluation, Occult offers genuine partial replication without any change. The only drawback is that clients might need to visit the master replica more often, in order to get a consistent version of the read object.

The compression of causal timestamps together with the genuine partial replication makes Occult a valid option for a causally consistent edge store. The only drawback is that clients do not have a *true* local replica, because writes must always be done to the master replica.

2.4.15 Saturn [15]

This is the second system to use a timestamp with a single part for the entire system. In this case, the metadata is a *label*, which works as a tag for updates. It is created in a *gear*, which exists in every server. The *label sink* is responsible for sending updates to other datacenters and they are received by a *remote proxy*. The clocks used in the labels are physical ones.

Clients track their causal history by keeping an updated label that is used in the operations. Saturn's new components handle all metadata, but the labels are stored alongside the object data.

If a client is reconnecting to a datacenter, it must firstly *attach* to it. It sends the previously seen label and the process ensures that the client's causal history is consistent with the datacenter's state. Sometimes, the client must wait for remote updates to arrive at the new datacenter.

For a read, the client only needs to send the object's key. The gear intercepts the request for recovering the label associated with the stored value. The client receives both and replaces its label if the returned one is newer.

Client writes have the object's key, new value and the client's label. The server's gear creates a new

label and stores it with the new value on the server's persistent storage. Then, the update data and metadata is sent to the label sink for being replicated. Finally, the new label is returned to the client, which replaces the old value.

The object replication separates the propagation of the object's data and label. The transmission of labels follows a well defined tree topology through *serializers*, but data can be delivery in any order. If a serializer does not have lower nodes that replicate the object associated with the arrived label, then it does not propagated it to them. This small detail gives genuine partial replication to Saturn.

Remote updates are only visible when the label arrives. The false dependencies are closely related with the arrival of the different labels, meaning that concurrent operations are executed in that order. If a label arrives before its corresponding data and the time to transfer all of it is high, other remote updates will be blocked. So, the label dissemination tree is built in order to minimize the difference in the times of arrival of the metadata and the data. Sometimes, it is necessary to introduce some delays in the transmission of labels, because they tend to be smaller and to arrive earlier.

When some serializer fails, datacenters will eventually become aware of it, changing to a different tree topology. The transition is made without stopping the system, but updates received from the new configuration are only applied after being sure that all other datacenters share the new tree. This is accomplished by using a special type of label, that signals the change of the tree for each datacenter.

To ease the *migration* of clients to other datacenters, there is another special type of label that is replicated to the new client's datacenter. It will ensure that all the causal history of the client is available at the destination.

Finally, this system fulfills the majority of the edge computing requirements. It supports genuine partial replication, the size of the metadata does not depend on the number of participants nor the number of objects, and it improves on the amount of the false dependencies. However, the complex topology does not allow a quick reconfiguration of the tree and the migration must use the origin datacenter for creating the special label, leading to an increase of this operation latency.

2.4.16 C^3 [16]

Like Saturn, this system separates the dissemination of data and metadata through two different layers (*storage* and *causality*, respectively) to provide (non-genuine) partial replication. However, it does not rely on the complex topology for propagating the metadata. So, there is the need for increasing the timestamp size to one part per datacenter. Each entry stores the logical clock of the corresponding datacenter.

Unlike all the other surveyed systems, clients do not store any metadata. Servers have a *local operations counter*, a *finished operations timestamp* and a *currently executing operations timestamp*.

On reads, clients provide the object key and get the associated value directly from the storage layer.

The causality layer is not contacted, because clients do not store system metadata.

When a client wishes to write to an object, it must send the key and the new value to the storage layer of its local datacenter. Then, the request is forwarded to the local causality layer, together with its unique identifier and the set of remote replicas that also store the object. The update dependencies are expressed by the currently executing operations timestamp. The local operations counter is incremented, used to tag the operation and replaces the datacenter entry in the currently executing operations timestamp. All this information is called a *label*. This label is added to a local log of pending operations and propagated to the object's remote replicas, that change the currently executing operations timestamp by taking the entrywise maximum between itself and the update dependencies. When all entries from the datacenter's finished operations timestamp are greater or equal to the update dependencies, the new value for the object is made visible at the storage layer. At the same time, the update's operation counter replaces the origin datacenter entry in the finished operations timestamp.

To migrate to a different replica, clients have a similar process to the one described for updates. The main difference is that only the currently executing operations timestamp is propagated to a single replica (*target datacenter*).

Due to the concurrency provided by the two multipart timestamps, remote update visibility is low. However, there is a significant impact in the local update latency, because the local replica may depend on foreign operations that are not yet applied.

When considering the edge computing paradigm, the only real problem with C^3 is the metadata size. The delay introduced before applying local updates is balanced by the lower remote update visibility. For collaborative work, this system might not be the most suitable alternative, given that two users connected to the same node would see updates with a considerable waiting time.

2.4.17 Comparison

In order to summarize the most important system characteristics, Table 2.1 classifies the the surveyed solutions according to the chosen technique, metadata size, amount of false dependencies and support for partial replication with its genuineness. In spite of enabling a comparison among the systems, Table 2.1 has too much detail. A more natural distribution of the solutions is on Figure 2.5, which is a plot comparing the different metadata sizes with the amount of false dependencies. The Δ at the end of some systems indicates the support for partial replication.

Figure 2.5 shows that a higher metadata size usually means a lower amount of false dependencies. Given that Okapi keeps an inter-datacenter synchronization, it has a more complex stabilization protocol and takes longer for remote updates to become visible. C^3 is on the same sector, because local updates may depend on remote ones that are not yet present, which delays their application in the original replica. Kronos and COPS have the same amount of metadata, but the second compresses the object's clock to

Systems	Technique	Metadata	False Dependencies	Partial Rep.
Kronos	Explicit check	Dep. graph O(K)	$\times$	$\times$
COPS	Explicit check	Keys' vector O(K)	E	$\times$
Orbe	Explicit check	Servers' vector O($N \times M$)	$E+S$	$\times$
PRACTI	Sequencer	Nodes' vector O(M)	$E+S$	Nodes$^{\times}$
Lazy Replication	Sequencer	Replicas' vector O(M)	$E+S$	$\times$
ChainReaction	Sequencer	DCs' vector O(M)	$E+S+ID$	$\times$
SwiftCloud	Sequencer	DCs' vector O(M)	$E+S+ID$	Client*
Legion	Sequencer	Peers' vector O(R)	$E+S$	Client$^{\times}$
Cure	Stabilization	DCs' vector O(M)	$E+S+ID$	$\times$
Okapi	Stabilization	DCs' vector O(M)	$E+S+ID+ED$	$\times$
Eunomia	Stabilization	DCs' vector O(M)	$E+S+ID$	$\times$
GentleRain	Stabilization	Scalar O(1)	$E+S+ID+ED$	$\times$
POCC	Optimistic	DCs' vector O(M)	$E+S+ID$	$\times$
Occult (no optimizations)	Optimistic	Partitions' vector O(N)	$E+S$	Server$^{\checkmark}$
Saturn	Tree	Scalar O(1)	$E+S+ID$	Server$^{\checkmark}$
C^3	Stabilization	DC's vector O(M)	$E+S+ID+ED$	Server$^{\times}$

Table 2.1: Comparison of the different solutions. K is the number of all objects in the system. N is the number of partitions. M is the number of replicas and R is the same, but per container. E, S, ID and ED represent inter-datacenter element, intra-server, intra-datacenter and inter-datacenter dependencies, respectively. $^{\checkmark}$ and $^{\times}$ stand for genuine and non-genuine partial replication, respectively. Regarding SwiftCloud's partial replication, there is a client cache that only replicates its interest set, but the datacenters have a full copy.

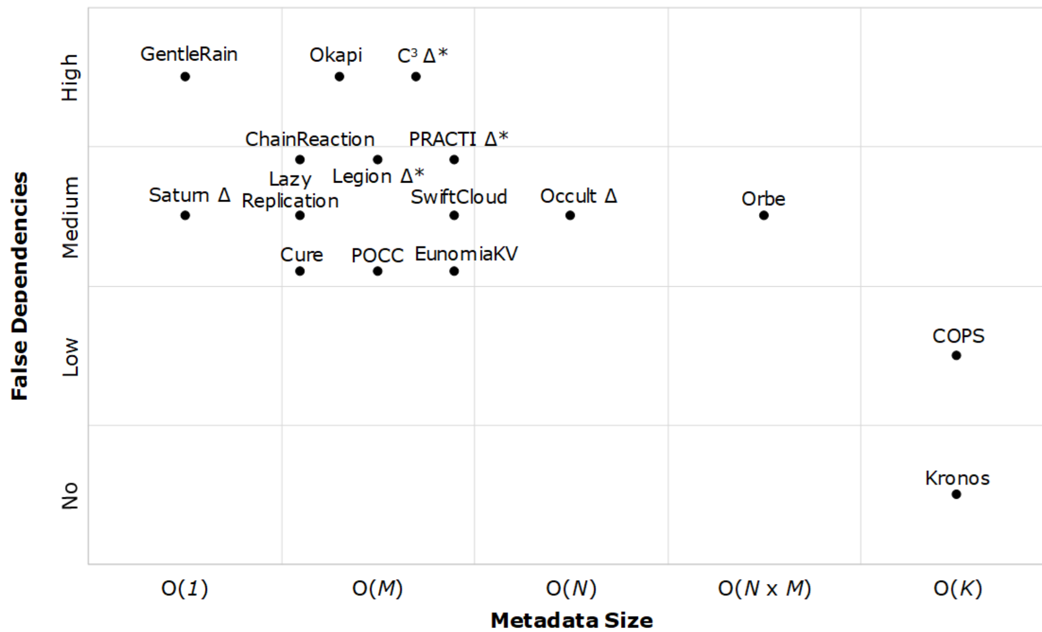

Figure 2.5: Graphic distribution of the systems according to their false dependencies and metadata size. Δ stands for genuine partial replication and Δ^* is for non-genuine. Low false dependencies stand for only one level on the table. Medium is either two or three. High is all of them.

only a scalar, while the first considers that operations are made at only one datacenter. Like GentleRain, Saturn keeps a constant metadata size, but is able to improve over the amount of false dependencies.

Regarding the different techniques on Table 2.1, there are five categories: explicit check, sequencer, stabilization, optimistic and tree. Explicit check exists when a server knows the objects that are missing to satisfy the dependencies (Kronos and COPS) or asks other specific entities about their current state (Orbe). Solutions that use a sequencer technique either have a log exchange replication (PRACTI, Lazy Replication, SwiftCloud and Legion) or a replication strategy with a well defined sequence (ChainReaction). Stabilization methods can be global, universal or site. Global stabilization uses a protocol involving the local datacenter partitions (Cure and GentleRain). Universal stabilization either does the same as global, but it is followed by an inter-datacenter protocol (Okapi), or datacenters account for remote operations before applying locally created updates (C^3). Site stabilization is done by a single entity of the datacenter, which is only responsible for the replication of updates (Eunomia). Optimistic mechanisms make clients responsible for guaranteeing the causality rules (POCC and Occult). The tree technique was developed for Saturn and is related to the dissemination of metadata.

Looking at partial replication, PRACTI, Legion, Occult, Saturn and C^3 are the only systems that may have nodes with a subset of the global state. From these, Occult and Saturn are the ones that offer genuine partial replication, meaning that nodes only get metadata regarding objects that they replicate. SwiftCloud is an exception, because it allows clients to cache a subset of all system objects, but all

datacenters have full replicas.

2.5 Limitations of Cloud Solutions on the Edge

Firstly, it is important to understand that all systems work in the conditions they were implemented. In spite of sharing a fair amount of characteristics, edge computing has some specific properties that do not belong to the cloud. So, the surveyed systems would not work if directly applied to the edge.

Looking at the different techniques to enforce causal consistency, some strategies are more suitable than others. For instance, the use of an explicit check means that the amount of metadata is dependent on the number of objects (Kronos and COPS) or that nodes need to exchange messages for inquiring about objects they do not replicate (Orbe). Both options do not scale to the size of the edge. Sequencer techniques are based on log exchange with repeated operations (PRACTI, Lazy Replication, SwiftCloud and Legion) or follow very strict replication strategies (ChainReaction). Stabilization creates an inter-node dependency for making remote updates visible (Cure, Okapi, Eunomia, GentleRain and C^3), which would be problematic with the higher edge churn.

The only two reasonable techniques are: optimistic and tree-based. The former is controlled by the client and may resort to waiting (POCC) or retrying (Occult), while the second delivers the updates after getting the corresponding label (Saturn). So, the node can decide what to do without consulting any third party.

For the timestamp size analysis, it must be a scalar. It is the only approach that complies with the unknowingly large size of the edge. In order to have the least amount of false dependencies, Saturn is the best option.

The majority of the solutions assumes full replication in the datacenter. As seen in Section 2.1, edge computing nodes do not have the same memory capacity. Regarding the partial replication techniques, the genuine approach is the most desirable, because nodes would only receive messages for objects of their interest. Furthermore, it would reduce the number of messages on the network. Occult and Saturn are the only ones to offer it.

In spite of being the system that fills most of the requirements, Saturn's optimization algorithm for building the dissemination tree is not the best strategy for the new paradigm. It does not cope with the much more dynamic environment, probably needing a high number of backup configurations. Furthermore, the amount of migrations to access locally unavailable objects is expected to increase and might take too long for some applications.

Considering all these issues, there is space for improvement on the edge computing paradigm. Current edge storage solutions only offer eventual consistency and causally consistent cloud stores make assumptions that are not valid on the network edge.

Summary

This chapter introduced edge computing and causal consistency. Usually, edge storage systems prefer weak consistency models, because they offer the lowest impact on the performance. However, they are difficult to reason with, sometimes leading to unexpected behavior. While there are many causally consistent cloud stores, they make a set of assumptions that do not hold at the edge of the network. The lack of storage and processing power, as well as the much higher number of devices are the main drawbacks to the direct use of cloud solutions on the edge.

In the next chapter, there is a proposal of a novel system to extend the existing cloud solutions, in order to include the requirements needed by the edge of the network. As it will explain, this system can grow independently of the number of participants or stored objects, while offering genuine partial replication.

3

Gesto: Geo-referenced Edge Store

Contents

3.1 Goals 38
3.2 Design 38
3.3 Multipart Timestamps 41
3.4 Protocols 42
3.5 Correctness 48
3.6 Reconfiguration and Faults 50

This chapter introduces Gesto, the first extension of causally consistent cloud stores to the network edge. Section 3.1 expresses the goals that need to be fulfilled. Section 3.2 overviews the design of Gesto, highlighting each component and their interaction. Section 3.3 describes the system metadata. Section 3.4 details the different protocols, which are proven to ensure causal consistency on Section 3.5. Finally, Section 3.6 addresses some techniques for adapting the system and additional measures to overcome faults.

3.1 Goals

Instead of proposing an universal system to enforce causal consistency over all networks, Gesto focuses on connecting edge devices to the nearest datacenter. The way datacenters interact to propagate updates among them can follow any of the surveyed solutions on Chapter 2.

For accomplishing it, Gesto must work as an extension of the causally consistent cloud storage services with small and constant metadata. However, the conversion of metadata from the edge to the cloud must require minimal changes, in order to minimize the impact on performance.

Given that clients are expected to be connected most of the time to nodes on the edge, the probability of the requested data not being locally available is non-negligible. This leads to clients requesting migrations more frequently. Gesto must support fast client migrations, without introducing inconsistencies in the state.

Finally, there are a considerable amount of edge devices that, in some point in time, could provide their resources for improving the performance perceived by the end-users. Furthermore, the large increase on the number of nodes also increases the frequency of maintenance on the overall system, which sometimes may result in nodes being removed from the network. Gesto must provide easy join procedures for new nodes and safe leave protocols for nodes that wish to exit. For the latter, the nodes must support the migration of their clients to other instances.

3.2 Design

This section starts with a small introduction of the different system components. Then, it overviews the client operations, together with an example of the sequence of steps that a client write follows.

3.2.1 System Components

For accomplishing the proposed goals, Gesto creates a three-tier architecture, as depicted in Figure 3.1. Overall, there are the four main components:

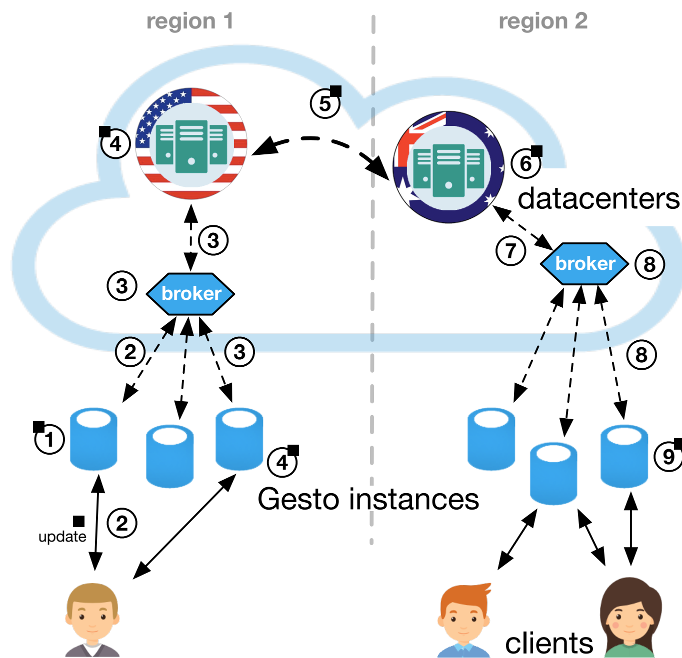

Figure 3.1: Gesto's three-layer architecture. The numbers represent the steps that an update issued at the leftmost Gesto instance of *region 1* will follow until being propagated to the rightmost Gesto instance of *region 2*. The execution flow does not depict the payload transmission within regions (done by 5). Steps with ■ indicate that they require the update's payload to be executed.

Cloud Storage Service. A causally consistent key-value storage system designed to run across a small number of datacenters. As in most surveyed systems, each datacenter replicates the whole application state and is also linearizable [42].

Regional Broker. A regional metadata management component in charge of orchestrating the metadata propagation within a region.

Gesto Instances. A partial replica of the application state (the state maintained in the cloud storage service instantiated by datacenters) that resides in the logical extreme of the network, typically one in each cloudlet (edge node). Each Gesto instance has a local datacenter to which it is connected, namely its parent datacenter.

Client Proxy. This component connects clients to Gesto. It forwards the client requests to the Gesto instances or the datacenter. Before sending them, these may be enriched with causal information that is used by Gesto to guarantee causal consistency.

3.2.2 Client Interaction

Clients interact with the system through a client proxy by performing read and write operations. These operations can be sent to any replica (a Gesto instance or a datacenter). However, for better perfor-

mance, clients should mostly interact with their closest replica. Before performing any operation, clients need to *attach* to a replica. The purpose of the attach procedure is to ensure that the target replica has a state that is consistent with the causal past of that client. Once attached, the client can issue read and write requests without further synchronization.

The process of detaching from a replica and attaching to another (for instance, to access a data item that is not cached in the origin replica) is called a *migration*. A client migration within a region is assumed to be a relatively infrequent operation (clients should perform multiple local reads/writes before migrating again), but it is not assumed to be rare. In fact, client migrations is a requirement that derives from the need for partial replication, the only realistic assumption in face of resource-constrained cloudlets.

In the following sequence, there is a description of the steps that are executed from the point a client issues a write request in a given region, up to the moment the update is applied not only in other replicas of the same region, but also in replicas of remote regions. Although the main concern is for mechanisms *inside* a region, Gesto is modular and can be combined with whatever mechanism the datacenter natively uses for maintaining causal consistency across regions. Thus, the example below also illustrates the interplay between Gesto and the datacenter's native causal consistency mechanisms. The execution flow, depicted in Figure 3.1, is as follows:

Step ① : The replica (a Gesto instance) handles a local write request by assigning it a multipart timestamp (its format is discussed in Section 3.3). Multipart timestamps (*MP_TS*) have enough information to order updates according to causality.

Step ② : The replica returns the *MP_TS* to the client, which assumes that the write is completed. Concurrently, the replica propagates the update payload to the regional replicas (including the parent datacenter) and the corresponding *MP_TS* to the regional broker.

Step ③ : When the regional broker receives the *MP_TS*, it processes it and sends it to the regional replicas that store the associated object, but not to the original one.

Step ④ : When a replica receives both the payload of the update and the corresponding *MP_TS* coming from the regional broker, it installs the update and makes it visible to local clients.

Step ⑤ : When the regional datacenter has made the update visible, it generates the metadata required by the native inter-datacenter causal consistency protocol. From the perspective of this protocol, the update installed by Gesto is handled as its own local update. Then, the update is propagated to other datacenters using the native protocol. Gesto is oblivious to the internals of the inter-datacenter replication mechanism. On the receiving side, there needs to be some changes on the update, before it is applied in the Gesto instance:

Step ⑥ : The native inter-datacenter protocol ensures that the update is received by the datacenter in causal order. Then, it generates a new Gesto *MP_TS* and locally installs the update.

Step ⑦ : The datacenter propagates the update payload to the regional replicas. Concurrently, it propagates the associated *MP_TS* to the regional broker.

Steps ⑧ and ⑨ : Steps ③ and ④ are equivalent, with the datacenter as the original replica.

By the previous example, it is possible to see that Gesto differentiates the network paths used to propagate the update payload and the corresponding multipart timestamp. It is agnostic to the process of propagating the update payload (different deployments can use different propagation strategies), because remote updates are applied by the order that metadata arrives. In the exposition, this is abstracted by calling a payload non-blocking multicast primitive whenever the content of updates needs to be shipped to other replicas. There are very few assumptions about this service, not even in the order of which updates arrive. Still, there is some interplay between the properties of the payload propagation service and causality, which is discussed in Section 3.6. Differently, multipart timestamps follow a specific path. This path and the changes performed to these timestamps during the propagation process are instrumental to keep their small size.

3.3 Multipart Timestamps

Gesto associates with each update a multipart timestamp, that is only meaningful within the region were it was generated and is not valid on any other. Clients also maintain a multipart timestamp as a summary of their causal past. This multipart timestamp allows the synchronization of the Gesto instance with the client on attach operations. So, the design of the multipart timestamps is key to guarantee fast and consistent intra-region client migrations.

A multipart timestamp includes two entries: a *local timestamp (lts)* and a *regional timestamp (rts)*. Each one of them is a $\langle src, clock \rangle$ tuple that includes a source field (*src*) for tracking which entity created the timestamp, and a clock field (*clock*), that captures the timestamp creation time.

The *lts* entries are created by the replica where the update is issued. Therefore, the *src* field indicates the origin replica of the update. In practice, clients use the *lts* to keep track of the most recent *local* update they have observed at the replica that they are attached. *A local timestamp can only be compared with other local timestamps created in exactly the same replica.*

Unlike the *lts*, *rts* entries are always assigned by the regional broker. Whenever the broker receives update metadata from some regional instance, its regional entry is changed to a new higher value. The *rts* is used by clients to keep track of causal dependencies for remotely created updates. *It is the regional timestamp that allows to order updates generated in different replicas of a given region.*

The use of a single regional entry for keeping track of all remote causal dependencies allows a reduction in the client metadata. Otherwise, it would have to store a different *lts* for each instance that exists in a region, which can total a very large number. Intuitively, Gesto keeps track of updates that

Algorithm 1 Client code.

```
    ▷ State kept by the client proxy
 1: client_mp_ts, pref_cloudlet
    ▷ Handles a read request
 2: function READ(key)
 3:     value ← ⊥
 4:     while value == ⊥ do
 5:         send READ(key) to pref_cloudlet
 6:         receive reply from pref_cloudlet
 7:         if reply == error ∨ reply == ⟨error, cache_miss⟩ then
 8:             pref_cloudlet ← PICK_NEW_REPLICA(key)
 9:             MIGRATE(pref_cloudlet)
10:         else
11:             ⟨value, mp_ts⟩ ← reply
12:             client_mp_ts.rts.clock ← MAX(client_mp_ts.rts.clock, mp_ts.rts.clock)
13:             if mp_ts.lts.src == pref_cloudlet then
14:                 client_mp_ts.lts.clock ← MAX(client_mp_ts.lts.clock, mp_ts.lts.clock)
15:     return value
    ▷ Handles a write request
16: function WRITE(key, value)
17:     uid ← GENERATE_ID( )
18:     mp_ts ← ⊥
19:     while mp_ts == ⊥ do
20:         send WRITE(uid, key, value, client_mp_ts) to pref_cloudlet
21:         receive reply from pref_cloudlet
22:         if reply == error then
23:             pref_cloudlet ← PICK_NEW_REPLICA(key)
24:             MIGRATE(pref_cloudlet)
25:         else
26:             mp_ts ← reply
27:             client_mp_ts ← mp_ts
28:     return ok
    ▷ Handles an attach request
29: function MIGRATE(target)
        ▷ The snapshot and attach requests are parallelized
30:     send SNAPSHOT(client_mp_ts, target) to pref_cloudlet
31:     send ATTACH(client_mp_ts, pref_cloudlet) to target
32:     receive mp_ts from target
33:     pref_cloudlet ← target
34:     client_mp_ts ← mp_ts
35:     return ok
```

happen in the local instance with more accuracy than for updates that happen in remote instances, given that the broker *merges* all remote updates in a single entry. The rational for this asymmetry is that clients are expected to perform much more local operations than remote ones.

3.4 Protocols

In this section, there is a detailed description of Gesto's protocols, accompanied by Algorithms 1 to 3, which are referenced throughout the text. The algorithms show the pseudocode for the protocols that run in the client proxy, Gesto instance and regional broker, respectively.

3.4.1 Reads and Writes

Clients read and write objects to their preferred replica, the one they are attached to. They can change their preferred replica by migrating to other replicas of the same region, which is explained in Section 3.4.3. The client proxy interacts with Gesto while keeping two pieces of state: *pref_cloudlet* identifies the client's preferred replica and *client_mp_ts* is a multipart timestamp that expresses the client's past.

Reads. Gesto's read protocol executes without any synchronization, as it is expected to be the most common operation. This is possible, because the migration procedure ensures that clients can only perform operations when the local instance is consistent with their state. Thus, the client reads are not required to forward its own *client_mp_ts* (Algorithm 1, line 5). The replica handles the request by returning the stored value together with its corresponding timestamp (Algorithm 2, line 5). However, if the replica does not cache the target key, an error is raised (Algorithm 2, line 7). In turn, the client incorporates the update (together with the update's causal dependencies) in its causal past by merging its *client_mp_ts* with the returned timestamp. This process is done by picking the regional entry with the greatest clock (Algorithm 1, line 12) and, if the update was generated in the same replica, the largest local entry (Algorithm 1, line 14).

Writes. On writes, the client starts by generating an unique identifier for the update (Algorithm 1, line 17). Then, it simply forwards the update request to its preferred replica, along with the *client_mp_ts* (Algorithm 1, line 20). The receiving replica assigns a new multipart timestamp to the update (Algorithm 2, line 9), by preserving the *rts* and replacing the *lts* clock with a value that is higher than all of the previous ones assigned to any local update. This guarantees that the timestamp assigned is greater than the timestamps associated to any operation observed by the client, which is key to guarantee consistent client migrations. If the key is locally cached, the update is applied to the local key-value store (Algorithm 2, line 11). Then, the metadata associated with the update is sent to the regional broker (Algorithm 2, line 12), while the payload goes to all other regional replicas (Algorithm 2, line 14), including the parent datacenter (as noted before, Gesto is agnostic to the algorithm used to support payload propagation). The new timestamp is returned to the client (Algorithm 2, line 15). Finally, the client incorporates the new update into its causal past, by simply overwriting its old *client_mp_ts* with the newly created timestamp (Algorithm 1, line 27).

3.4.2 Intra-regional Update Replication

As mentioned, Gesto decouples the dissemination of data and metadata. In order to apply a remote update, a replica must have received both the associated payload and the metadata. The payload is directly received from the originating replica. Payloads can be received out of order and are temporarily stored in a *Payloads* buffer until they can be applied in causal order (Algorithm 2, line 36). Metadata is

received in an order that respects causality. Therefore, updates are applied in the exact order in which metadata is received (Algorithm 2, line 28). This is illustrated in function INTRA_META in Algorithm 2, line 26.

Firstly, metadata is propagated from the origin replica to the regional broker (Algorithm 2, line 12). The broker *merges* the metadata produced in different replicas in a single stream, consistent with causality, and then propagates it to the remaining regional replicas (Algorithm 3, line 7). For each replica, the broker only sends the metadata associated to updates on items that are cached on that replica, filtering out the rest. This enhances performance, because it frees replicas from processing irrelevant metadata and reduces the network traffic.

To simplify the process of creating a causally consistent stream at the broker, there is the need for FIFO channels between the broker and the replicas. Moreover, replicas must send multipart timestamps to the broker in local timestamp order. Under these constraints, the stream is causally consistent if the broker assigns regional timestamps respecting the order by which it receives multipart timestamps from replicas. If there are multipart timestamps from different replicas that have not yet been processed by the broker, they are necessarily concurrent and can be serialized in any order.

For instance, update b was originated at replica 1, it can only depend on update a from replica 2 if a was visible at 1 before b was issued, such that the client that issued b could have read a. Therefore, the metadata associated to a must have been received by the broker before b's metadata.

When merging multipart timestamps coming from different regions into a single stream, the broker updates the regional entry of those timestamps (Algorithm 3, line 4). The new value of the regional entry captures the serialization of the updates received from an instance with regard to concurrent updates that have or will be received from other instances (or even, remote regions). This is achieved by simply assigning a monotonically increasing clock to regional entries.

There are a few subtle issues that are worth mentioning. Because the client can submit the same write to multiple siblings (for instance, if it times out when issuing the request to a replica), an update can have more than one multipart timestamp (created by different instances). Still, the update is locally applied only once, when the first multipart timestamp is received (Algorithm 2, line 27). Also, clocks are loosely synchronized, which enables the local application of a remote update with a timestamp clock value that is greater than the local wall clock. To ensure that clock values match the causal order, the replica may have to wait for its wall clock to catch up when issuing new timestamps (Algorithm 2, line 38).

3.4.3 Client Migration

A migration is the procedure that allows a client to attach to a new replica (the *target* replica) after being previously attached to another replica (the *origin* replica). A client may migrate if: its current preferred replica does not replicate a data item, its preferred replica becomes unreachable, or it has physically

Algorithm 2 Gesto instance code (*cloudlet*).

```
    ▷ State kept by cloudlet
 1: Payloads, Applied, MaxRegional, LastFrom[], cloudlet, broker, this_region
    ▷ Handles a read request
 2: function READ(key) from client
 3:     if CACHES(cloudlet, key) then
 4:         ⟨value, mp_ts⟩ ← KV_GET(key)
 5:         return ⟨value, mp_ts⟩
 6:     else
 7:         return ⟨error, cache_miss⟩
    ▷ Handles an update request
 8: function WRITE(uid, key, value, past) from client
 9:     mp_ts ← GENERATE_MPTS(past)
10:     if cloudlet == datacenter then EXPORT_UPDATE(uid, key,value)
11:     if CACHES(cloudlet, key) then KV_PUT(key,⟨value, mp_ts⟩)
12:     send INTRA_META(uid,key, mp_ts) to broker
13:     replicas ← REGIONAL_REPLICAS(key,this_region) \ {cloudlet}
14:     multicast PAYLOAD(uid, key,value) to replicas
15:     return mp_ts
    ▷ Waits until is safe for a client to be attached
16: function ATTACH(past, origin) from client
17:     wait until MaxRegional ≥ past.rts.clock
18:     wait until LastFrom[origin].lts.clock ≥ past.lts.clock
19:     lts ← ⟨cloudlet, 0⟩
20:     rts ← ⟨broker,MAX(LastFrom[origin].rts.clock, past.rts.clock)⟩
21:     mp_ts ← ⟨lts, rts⟩
22:     return mp_ts
    ▷ Handles a snapshot request
23: function SNAPSHOT(past, target) from client
24:     mp_ts ← GENERATE_MPTS(past)
25:     send INTRA_META(⊥, faux(target), mp_ts) to broker
    ▷ Handles remote metadata coming from the broker
26: function INTRA_META(uid, key, mp_ts) from broker
27:     if uid ≠ ⊥ ∧ uid ∉ Applied then
28:         wait until ⟨uid, key, value⟩ ∈ Payloads
29:         KV_PUT(key, ⟨value, mp_ts⟩)
30:         Payloads ← Payloads \ {⟨uid, key, value⟩}
31:         Applied ← Applied ∪ {uid}
        ▷ Updates bookkeeping information
32:     MaxRegional ← mp_ts.rts.clock
33:     LastFrom[source] ← mp_ts
34:     return ok
    ▷ Handles an update payload coming from a replica
35: function PAYLOAD(uid, key, value) from replica
36:     Payloads ← Payloads ∪ {⟨uid, key, value⟩}
    ▷ Generates, based on the client's past, a new multipart timestamp
37: function GENERATE_MPTS(past)
38:     wait until CLOCK( ) > past.lts.clock
39:     lts ← ⟨cloudlet, CLOCK( )⟩
40:     rts ← past.rts
41:     return ⟨lts, rts⟩
```

moved and it is now is closer to other replica. When faced with a cache miss, a client may directly migrate to the local datacenter (that replicates all items) or, when this information is available, to another cloudlet that also replicates the desired data item. Migration is supported by an ATTACH operation, which is mandatory, and by a SNAPSHOT operation, which is optional and has the purpose of reducing the latency of the migration procedure.

Attach. In order to perform a migration, a client issues an ATTACH request to the target replica (Algo-

Algorithm 3 Regional broker code (*broker*).

▷ State kept by broker
1: *broker_clock*, *broker*, *this_region*
▷ Handles a new remote operation originated at a local cloudlet or datacenter
2: **function** INTRA_META(*uid*, *key*, *mp_ts*) from *sender*
3: *broker_clock* ← *broker_clock*+1
4: *mp_ts.rts* ← ⟨*broker*, *broker_clock*⟩
▷ Propagates to regional replicas
5: *replicas* ← REGIONAL_REPLICAS(*key*, *this_region*) \ {*sender*}
6: **for all** *r* ∈ *replicas* **do**
7: **send** INTRA_META(*uid*, *key*, *mp_ts*) **to** *r*

rithm 1, line 31). The client's multipart timestamp is sent as a parameter, because it captures its causal past. The local entry keeps track of all operations in the client's past that have been issued on the origin replica. The regional entry keeps track of all operations in the client past that have been issued on a remote replica. In order to successfully attach to a target replica, the client must wait until that replica has applied all those past operations. To track which updates have already been locally applied, each replica maintains some bookkeeping information about their own region. Namely, a replica keeps track of the last update it has applied from each remote replica (vector *LastFrom*) and of the highest regional timestamp ever seen (*MaxRegional*). Thus, a replica can satisfy an attachment request as soon as *LastFrom[origin]* and *MaxRegional* are, respectively, higher or equal to the local and regional entries of the client's multipart timestamp (Algorithm 2, lines 17 and 18).

The second part of the attachment procedure consists on generating a new multipart timestamp that allows the client to interact with the target replica. Since the client has not yet observed any state on the new replica, the clock of the local entry of the multipart timestamp is simply assigned with value 0 (Algorithm 2, line 19). The clock on the regional entry is set to the most recent update that has been previously observed by the client. For this, the target replica checks the regional timestamp that was assigned by the broker to the last operation performed at the origin replica and that was received at the target replica, and compares it to the last operation observed by the client from other replicas. The assigned regional entry is the one with the highest clock (Algorithm 2, line 20). This conversion must be done at the target replica, because clients only keep track of the local clocks regarding updates performed at the origin and not of the regional clocks assigned to those updates.

Snapshot. The SNAPSHOT operation is executed at the origin replica (Algorithm 1, line 30). In spite of being optional, this operation may speed-up the attachment at the target replica. Assume that a client reads some object x on the origin replica with timestamp t_x, which makes it have t_x in the local part of its timestamp. When the client attaches to a target replica, it will need to wait until the target replica receives a timestamp greater or equal to t_x from the origin replica. Unfortunately, it may happen that the target instance does not cache object x. In fact, the target replica may cache very few items that are also cached at the origin replica. Thus, a higher timestamp from the origin replica may take an arbitrary amount of time to reach the target replica. The SNAPSHOT operation aims at fasten the synchronization.

To force an update of the metadata at the target instance, SNAPSHOT is implemented by emulating an update that is performed at the origin replica and that needs to reach the target replica, thus prompting the propagation of the associated metadata (Algorithm 2, line 25). To avoid incurring the full cost of updating a data object, SNAPSHOT is implemented as an update on a *faux* object that consumes no state. Updates on faux objects have no unique identifiers and do not require the exchange of payload messages; they only trigger metadata updates on the origin replica, regional broker, and target replica. There is a faux object for every pair of replicas.

Liveness. Since SNAPSHOT operations are not mandatory, there needs to be a mechanism to ensure that migrations complete in bounded time, even when the objects that are replicated at the origin and target replicas are not updated for long periods. Thus, in absence of further updates, each replica periodically executes a spontaneous SNAPSHOT operation. This bounds the migration latency and guarantees that, if no failures occur, a client migration always completes.

3.4.4 Inter-regional Update Replication

By design, Gesto is only concerned with keeping track of causality within a region and relies on any pre-existing protocol that is natively supported by the datacenter for inter-regional replication. This allows Gesto to be plugged into an existing system, in order to augment it with support for edge computing with minimal changes to the inter-datacenter replication protocols already in place.

The inter-regional replication works as follows. When a datacenter installs a new update, either issued by a local client or originating at a regional replica, it generates whatever metadata is required by the native inter-regional protocol. Then, the datacenter ships the update, together with the inter-regional metadata, to remote datacenters. Note that Gesto is oblivious to the structure of the inter-regional metadata and to how it is generated. Similarly, Gesto's multipart timestamp does not need to be shipped to remote datacenters and can be ignored by the native protocol. This ensures a modular design.

Eventually, a receiving datacenter will decide that it is safe to include the update on its state. So, it generates a fresh Gesto multipart timestamp (that it is only valid in its own region). This multipart timestamp contains a local entry that is greater than any local timestamp ever assigned by the datacenter and a regional entry that is set to 0. This is enough to guarantee that any client who reads this update from the datacenter can safely migrate to other regional replica. Finally, the datacenter acts as if the update was issued by a local client: installs the update, alongside its multipart timestamp; propagates the payload to all regional replicas; sends the multipart timestamp to the regional broker. The broker processes the metadata as if it was an update from its region and propagates it to the regional replicas, which, in turn, make the remote update visible to local clients.

3.5 Correctness

In this section, there are two proofs to show that the previous algorithms ensure causal consistency. They make some assumptions regarding the operation of each replica and the operation of the network that connects the replicas. Firstly, each replica is linearizable. Secondly, the channels between a replica and the regional broker are FIFO. The following theorems are enough for ensuring that Gesto enforces causal consistency.

Theorem 1. *Let client c be attached to instance i and let k be the key of a data item that is cached at i. If c has read update u on k (u belongs to c's causal past), then update u has already been applied to i.*

Proof. Let $\{\langle broker, urclock\rangle, \langle usrc, ulclock\rangle\}$ be the multipart timestamp associated with update u, after its metadata has been processed by the regional broker. There are three cases:

case 1) Client c has read u on instance i. In this case, from the assumption that instances are linearizable, if c has read u, u has been previously applied at i.

case 2) Client c has read u on instance $j \neq i$ and $j \neq usrc$. In this case, after c reads u, its metadata *client_mp_ts*$_c$*.rts.clock* $\geq urclock$ (Algorithm 1, line 12). Further updates to *client_mp_ts*$_c$*.rts.clock* can only increase its value (Algorithm 1, line 27 and Algorithm 2, line 40). Also, even if the client migrates to some other instance k before migrating to i, migration never decreases the value of *client_mp_ts*$_c$*.rts.clock* (Algorithm 2, line 20). Thus, when client c finally migrates to instance i, it still has *client_mp_ts*$_c$*.rts.clock* $\geq urclock$. Migration only terminates when the update timestamped with *client_mp_ts*$_c$*.rts.clock*, or newer, is applied at instance i (Algorithm 2, line 17). So, u has been applied on i when c becomes attached to i.

case 3) Client c has read u on instance $j \neq i$ and $j = usrc$. In this case, after c reads u, its metadata *client_mp_ts*$_c$*.lts.clock* $\geq ulclock$ (Algorithm 1, line 14). While the client is attached to j, further updates to *client_mp_ts*$_c$*.lts.clock* can only increase its value (Algorithm 1, line 27 and Algorithm 2, lines 38 and 39). Assuming that the client eventually migrates to i, there are two cases:

- The client migrates directly from j to i. In this scenario, migration only terminates when i has installed the set of all updates local to j whose *lts.clock* $\leq$ *client_mp_ts*$_c$*.lts.clock* (Algorithm 2, line 18). This set includes u.

- The client migrates to n and then to i. When the client attaches to n, the replica ensures that the client's new *client_mp_ts*$_c$*.rts.clock* $\geq urclock$. This is guaranteed, because migration: waits until i has installed the set of all updates local to j whose *lts.clock* $\leq$ *client_mp_ts*$_c$*.lts.clock*, which includes u (Algorithm 2, line 18); and sets the client's new *rts* to be at least equal to the maximum *rts* observed by updates local to j (Algorithm 2, line 20). This ensures that *client_mp_ts*$_c$*.rts.clock*

$\geq urclock$. Therefore, by case 2, the client will only attach to i when u has been applied, even if there are more migrations in between.

$\square$

Theorem 2. *Let u be an update on a data item with key k and u' be an update on a data item with key k' such that $u' \rightarrow u$ (k may be k' as well as any other key). Let i be an instance that caches both k and k'. If u has been applied on i, then u' has been previously applied on i.*

Proof. Let $\{\langle broker, urclock\rangle, \langle usrc, ulclock\rangle\}$ be the multipart timestamp associated with update u and $\{\langle broker, urclock'\rangle, \langle usrc', ulclock'\rangle\}$ be the multipart timestamp associated with update u', after the regional broker processed the metadata.

Throughout the proof, the goal is to show that the regional broker always observes u' before u, and therefore serializes u' before u. Since the regional broker dictates the order in which remote updates are made visible at a given replica, this guarantees that all replicas install u' before u. Like in the previous proof, there are three cases:

case 1) Both updates u and u' are local to the same replica i. There are a few subcases to consider:

- A client reads u' from i and does not migrate before issuing u. The protocol guarantees that $ulclock > ulclock'$, because, after reading u', *client_mp_ts$_c$.lts.clock* $\geq ulclock'$ (Algorithm 1, line 12) and the protocol ensures that $ulclock \geq$ *client_mp_ts$_c$.lts.clock* (Algorithm 1, line 27 and Algorithm 2, lines 38 and 39). Since the regional broker receives timestamps from a replica in *lts* order, this guarantees that the broker will serialize u' before u.
- A client reads u' at i, but migrates to n before coming back to i for issuing u. By Theorem 1, u' will be visible at n when the client attaches. Thus, u' must have already been observed by the broker. Therefore, any future update issued by the client is serialized after u'.
- A client reads u' in another replica j, then migrates to i and issues u. This implies that u' was visible at j before u was issued. Thus, the broker will serialize u' before u.

case 2) u' and u are local to different replicas (i and j, respectively) of the same region. There are a few subcases to consider:

- The client that issues u reads u' in replica $n \neq i$. This implies that u' has already been serialized by the regional broker before being visible at n. Therefore, u is inevitably serialized after u'.
- The client that issues u reads u' from i (the origin replica of u'). Since u is local to some other replica j, the client has to migrate after reading u' at i. By Theorem 1, when the client attaches to j, u' will be visible there. This implies that u' has already been serialized by the regional broker. Therefore, u is inevitably serialized after u'.

case 3) u' and u are local to replicas of different regions (r_1 and r_2, respectively). As previously addressed, the propagation among regions is done by datacenters. Thus, since a datacenter only ships updates to other regions once these have been locally installed, when dc_{r_1} receives u from dc_{r_2}, u' is already installed. To show that this invariant is maintained at other regional replicas, there are two more subcases to consider:

- u' was generated at replica i, which is not the regional datacenter. Before u' is processed at dc_{r_1}, the regional broker has already serialized it. At some point, u is received at dc_{r_1} and propagated to the regional broker, that serializes it after u'.
- u' was generated at the regional datacenter (dc_{r_1}). In this scenario, dc_{r_1} may receive u before shipping u' to the regional broker. Gesto guarantees that u' is received at the regional broker before u, by assigning a fresh local entry to u such that $ulclock > ulclock'$. Since the broker receives timestamps from each replica in local entry order, this guarantees that it serializes u' before u.

□

3.6 Reconfiguration and Faults

Given the dynamic network, Gesto must support some reconfiguration strategies for reacting to the changes. There are three actions: change on the replication set of a data item; adding a Gesto instance; removing a Gesto instance. In order to offer these actions, Gesto has a *directory service* that stores the most up-to-date distribution of objects across the regional replicas. Finally, Gesto also considers the existence of faults.

3.6.1 Directory Service

So far, there has been the assumption that the replication set was immutable, allowing all replicas and the broker to know precisely where each data item is replicated. However, the addition of reconfiguration actions to Gesto requires dynamic updates to this information. Thus, Gesto integrates an external directory service that registers which instances replicate each key. It integrates the following features, which are also included in widely-used services, such as ZooKeeper [43]. Firstly, this service is highly available and strongly consistent, which can be achieved by using a standard Paxos-based [44] replication mechanism. Secondly, the directory is versioned and all of its changes can be serialized.

The following mechanisms are independent of the policies used to change the replica membership and work correctly as long as all replicas observe the same sequence of membership changes. Also, most systems use course grain partitions as the replication granularity. Thus, in most deployments, the

number of partitions whose membership needs to be tracked is several orders of magnitude smaller than the number of individual items.

3.6.2 Changing the Replication Set

Removing a replica from a replication set is trivial. The replica that whishes to stop caching the data item starts by flagging the target key as invalid and ignores any further updates or metadata associated with that key. When clients try to read it, the replica returns a *cache miss* error. Then, it updates the directory service, such that, eventually, the broker and other regional replicas stop forwarding updates for that key to that instance.

Adding a new replica of a given data item is slightly more complex. Firstly, the directory service needs to be updated, such that the broker and the remaining replicas become aware of the new replica. However, due to the distributed nature of the architecture and the asynchrony of the system, when an update is generated at a given replica, the membership of the key registered in the directory service can be in some version i and, when the update is processed by the broker, the membership can be in some other version j. This can create scenarios where the new replica receives the payload for an update, but not the corresponding metadata, or receives the metadata, but never the payload. To circumvent this problem, when an update is processed at a given replica, the version of the replication set is recorded and sent to the broker, which is able to send the metadata to the same set of replicas that will receive the payload. Also, if a replica processes an update using version i, it ensures that future updates will use a version at least equal to i.

Finally, the new replica needs to get the state associated with the key. In many systems, updates overwrite the entire value associated with a key. In that case, no special state transfer mechanism needs to be put into place: the first update received by the new replica initiates the state for that key. However, it may happen that updates to a key are infrequent or incremental. In such cases, it is required to trigger a *state transfer*. Gesto implements state transfer as follows. The new replica picks any replica that already caches the desired data item and requests that a special *fork* operation is issued. The purpose of the fork operation is to obtain a current copy of the data item's state. Fork combines features of both read and write operations. It reads an object and gets its entire state (as any read operation) and, atomically, generates an update multipart timestamp *as if* the object was written, without changing its value. The state of the object, returned by fork, is used to ship the actual content of the object as an update to solely the joining instance. The multipart timestamp corresponding to the fork is propagated to the new replica, via the broker, as any other multipart timestamp. The joining node uses the fork as the first update to the cached object, allowing future reads, updates, and remote updates to its key.

3.6.3 Adding and Removing Instances

Adding Gesto instances to the system is trivial. They just need to execute the protocol described in the previous section to start replicating keys.

The process of gracefully removing an instance is executed as follows. The instance stops accepting new clients and returns cache miss errors to requests coming from the attached clients. In the meantime, the instance helps those clients in the migration to other replicas, by serving SNAPSHOT requests. When all clients have migrated, the instance updates the directory service and can then safely shutdown.

3.6.4 Faults

Faults may happen at several system components: on individual data nodes, on the servers that implement the regional broker or disconnections of entire cloudlets. For individual data nodes, the native replication scheme of the key-value store used at each cloudlet prevents the loss of data. The regional broker is a key component of Gesto's architecture and needs to be highly available. Standard Paxos-based techniques to build a replicated, highly available, regional broker are well understood today, so one of them can be used to solve this issue. The loss of a cloudlet can be addressed by using either optimistic or pessimistic schemes, as discussed below.

The pessimistic approach consists on using a *k-stability* mechanism, similar to the one from SwiftCloud [7], one of the surveyed systems. It guarantees that if a replica fails unexpectedly, any local update was either replicated somewhere else, or never exposed to clients outside the originating instance. Thus, clients local to the failing instance are able to safely attach to other ones, as it is guaranteed that any of their causal dependencies are elsewhere and will eventually become replicated. Unfortunately, this strategy has a cost: it increases the update visibility latencies, but not the latency of write operations, as a replica still returns as soon as it locally installs the update. So, for k-stability to work, clients are required to cache their write operations until these are k-stable. If a failure occurs, the client can migrate and re-issue the write operations that its previous instance was not able to propagate.

The optimistic approach trades availability for speed, by making the updates visible before they are copied to other replicas. If an instance fails, there are two scenarios. If all updates that have been accepted by the instance have been shipped and the corresponding multipart timestamps have reached the regional broker, nothing needs to be done, as the mechanisms used in steady state ensure that other instances will become consistent. If the instance fails, but some updates/metadata are lost, the client has two alternatives: it either blocks until the instance recovers, in order to not violate causality; or it cleans its causal past and re-attaches to another instance as a new client. The latter guarantees progress, but causal consistency may be violated.

Summary

This chapter has addressed the design of Gesto, an extension to causally consistent cloud stores. The separation between the propagation of the update payload and its multipart timestamp enables a new lightweight change to the metadata structure, with a small increase in its size. The additional information has a constant size, independent of the number of replicas or objects. Given the strategically placed broker, the metadata is enhanced with a compressed regional timestamp that uncovers the client dependencies outside the local instance. The novel metadata structure is instrumental in freeing clients from sticky sessions, allowing a never before seen mobility among the replicas. The next chapter explains a framework for edge stores, which is able to ease the implementation of the systems that are going to be evaluated. It also highlights specific changes that make each system unique.

4

Implementation

Contents

4.1 Framework for Edge Storage 55

4.2 Implementing Different Systems 58

This chapter addresses the implementation of Gesto, as well as some of the surveyed solutions. Section 4.1 presents the framework that guided the development of the systems, highlighting all of its components. Section 4.2 describes the two considered organizations and details each system's implementation decisions.

4.1 Framework for Edge Storage

In order to assess if Gesto is the best option for the network edge, it needs to be compared to other systems that employ different techniques. The chosen ones were: COPS [2], for offering the lowest amount of false dependencies; Occult [14], because it is an optimistic scheme and its design offers simple operations with flexible topologies; Saturn [15], given that it shares some of the strategies used in Gesto, but operates in more complex topologies. In addition to the causally consistent systems, Gesto is also compared with a system that only offers eventual consistency. A priori, eventual consistency should offer the best latency, because it makes updates visible as soon as they are received.

Thus, there are five systems that need to be implemented. To make the comparison fair, their implementation uses a common framework (same programming language, same storage subsystem, but different algorithms). In particular, when two systems require a given component, they should use the same implementation of such component. Therefore, the common framework considers the following components: *clients*, which are in charge of interacting with the system; *client receivers*, that get the client requests and redirect them to the right internal component; *partitions*, that store the replica state and reply to read/write requests; *timestamp senders*, that export update metadata for their replication; *timestamp receivers*, that get the remote update metadata and manage their application; *brokers*, that create a network among regional replicas. The last three components are only used by the approaches that propagate the payload separately from the metadata.

The framework was developed on Linux, using Erlang/OTP as the programming language. Each node executes the Erlang R16B02 virtual machine. Note that more recent releases of the virtual machine exist (at the moment of writing, the latest stable release is version 21), but this exact virtual machine was needed for running Basho Bench [45], a benchmarking tool that helps in the evaluation of the systems. All communication is done through Erlang's generic server behavior calls and casts.

The implementations consider the existence of a single region. The datacenter can be seen as any other instance, with the difference of storing all the available objects.

4.1.1 Client

Clients can perform three operations: read, write and migrate. Each operation is implemented as a different call in the Basho Bench's driver, which is used to assign specific behaviors. Alongside the driver,

there is a configuration file that, among other parameters, allows to specify the following settings: the chosen driver, the amount of clients, the duration of the test, the report interval and the operation frequency. The last one is what defines the behavior by restricting the global proportion of each operation. When an operation ends, the following one is randomly chosen, but the tool approximates the frequency of the run to the ones defined in the configuration.

Although Basho Bench gives some statistics regarding the client throughput and each operation latency, it lacks some information that is relevant for the evaluation. For instance, it does not output all the latency percentiles. Instead, it gives the median, ninety fifth, ninety ninth and maximum percentiles. So, this tool had to be changed, in order to obtain all the data for the evaluation. For finding the ninetieth percentile, its computation was added to a Basho dependency, called Bear. It is responsible for ordering the received latencies, requiring just the addition of two lines of code for the percentile to be available through Basho. However, the evaluation of the migration latency needs to get all percentiles. To minimize the impact, they are directly extracted from Basho, by storing all relevant values and performing the necessary computations when the results need to be reported.

On Chapter 5, there are experiments that analyze the impact of the client access pattern, while maintaining the same read/write ratio. This is ensured by a custom driver that has a single multi-step operation, composed by a sequence of reads and followed by a write to either a fully replicated data partition or to any of the partially replicated ones.

4.1.2 Client Receiver

This component is the entry point for the client requests. It hides the internal structure of the replicas, being responsible for forwarding the request to the local partition that might replicate the object or to the timestamp receiver for completing an attach. For knowing the correct partition, it uses a hashing function provided by Riak KV [46].

The indirection introduced by the client receiver may be extended with a load-balancing mechanism, that aims at reducing the load on heavily requested partitions. For example, if the most requested objects all fall in the same partition, it might be interesting to re-partition the data such that hotspots become located on different partitions. Such re-partitioning could reduce the latency perceived by the clients. In spite of being a possibility, the current prototype tries to remain simple and does not implement load-balancing.

4.1.3 Partition

Partitions store the object versions, making them the target of read and write operations. The client receiver forwards these requests, which are handled in different manners. On reads, the partitions

immediately return with either the value and, for causally consistent systems, the metadata associated with the requested key or a cache miss error. Writes are a little bit more complex. This component might need to create new metadata to tag the update, storing all information in the built-in term storage of the Erlang virtual machine. When exporting updates, all solutions contact the destination's sibling partition and the ones that separate the payload from the metadata forward the latter to the timestamp sender, which is explained in the following section.

Partitions also apply remote updates to the local replica state. If the payload arrives alongside the metadata, it is able to solely decide when it is safe to make the update visible. Otherwise, there is a synchronization between the partition and the timestamp receiver, with the second dictating the order in which updates are applied.

4.1.4 Timestamp Sender

When propagating the updates to other replicas, this module handles the transfer of their metadata to the broker. Similarly to Eunomia [11], it receives the metadata from all partitions and applies the site stabilization every millisecond. The higher frequency is necessary for coping with the proximity among replicas. If the interval increases, the remote update visibility latency is heavily impacted, because the latency among nodes is one order of magnitude lower than in the cloud scenario.

Given that the site stabilization runs after a period of time, the pending multipart timestamps tend to accumulate. To increase the efficiency on their propagation to the broker, the stable multipart timestamps are sent as a batch.

4.1.5 Timestamp Receiver

The timestamp receiver has two different, but equally important roles. Firstly, it is key in the application of remote updates to the local replica state. Secondly, it stores information for supporting client migrations.

After receiving metadata from the broker, the timestamp receiver delivers it at the partition that stores the associated object. Until the partition acknowledges the application of the update, every new update metadata that arrives at the timestamp receiver is queued. Eventually, the acknowledgement reaches the timestamp receiver, which takes the oldest metadata from the queue and delivers it at the right partition, repeating the previous process.

During the previous protocol, this component keeps information about the received metadata. At least, it stores the last received metadata from every other regional replica. The goal is to check if the local state is consistent with the client's causal history, allowing the attachment of new clients.

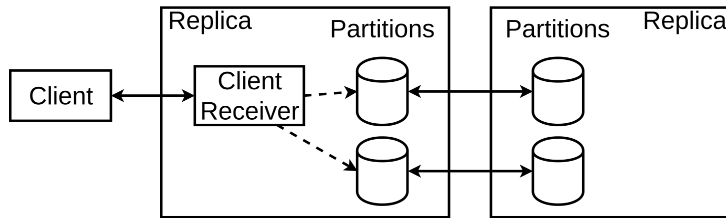

Figure 4.1: Interaction of the different components for systems that propagate the payload alongside the metadata.

4.1.6 Broker

Even though the broker has a valuable role in the propagation of remote updates, it keeps a simple functionality. It receives metadata from a replica's timestamp sender and forwards it to the timestamp receiver of other replicas or to the internal network of brokers.

After receiving a metadata batch, the broker goes over each one and puts them at the end of the outgoing queue to the nodes that replicate the associated object or to the broker that reaches those nodes. If there is metadata for allowing migrations (either Gesto's faux updates or Saturn's migration labels), it follows the path that leads to only the target replica. Every millisecond, the queues are flushed, allowing the metadata to reach its final destination.

4.2 Implementing Different Systems

As previously mentioned, there are two main approaches for propagating the data payload and the associated metadata. One consists in propagating the update's payload alongside the metadata, as illustrated in Figure 4.1. The other separates the propagation of the update payload and metadata, as depicted in Figure 4.2. Systems that use the former approach do not support an explicit migration operation (migration is implicit and can be inferred from the context of the operation).

The second layout can be seen as an extension of the first one, because there are several components that are always used regardless of how metadata is managed. For instance, partitions are able to communicate among themselves for the exchange of update payloads. However, in systems that decouple metadata propagation, an additional set of servers is needed for providing the support.

The following sections detail the implementation of each evaluated system, based in the common framework. They pinpoint the changes to specific components, in order to create the necessary behavior for each particular system.

4.2.1 Eventual Consistency

Eventual consistency implements the layout in Figure 4.1. However, there are some components that can be optimized, because of the weaker constraints that need to be preserved.

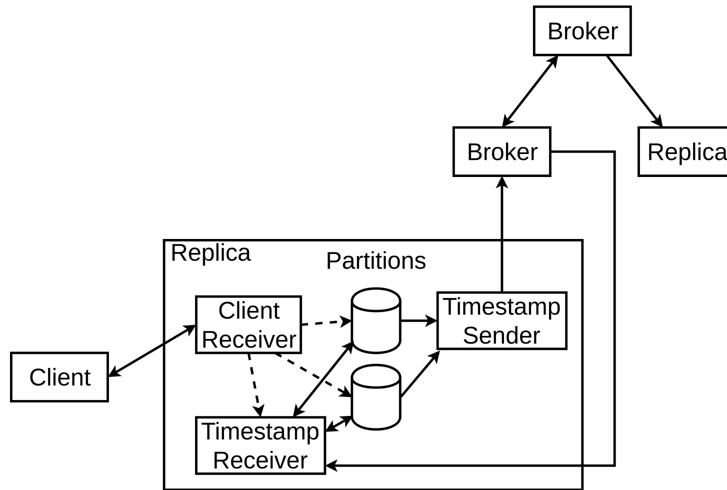

Figure 4.2: Interaction of the different components for systems that propagate the payload separately from the metadata.

Firstly, clients do not have to maintain any metadata. Write requests just need the object's key and the new value. The response serves just as an acknowledgement of the completion of the operation, without containing any additional information. Partitions do not apply dependency checks, storing the value with the highest physical clock. As soon as they get the new object value, they can make it visible.

4.2.2 COPS

COPS also uses the layout presented in Figure 4.1. Originally, COPS does not support partial replication, because of the way metadata is compressed after writes. For enabling a fair comparison among all solutions, COPS was extended to offer it.

If the write is made to an object that is replicated in every instance, the original compression strategy is still valid. So, the deployment assumes the existence of a particular object for allowing the reduction of the metadata size. For the remaining objects, the client does not compress the dependencies and appends the newly written version, which will prevent inconsistencies in the replication to other instances. As a small optimization, clients store their dependencies in a dictionary, which filters out different versions of the same object key.

Although client writes have the list of dependencies, they are not checked in the partition that receives the request. This comes from the assumption that each replica is linearizable. Before being able to apply a new remote update, the target partition sends a dependency check request for each update dependency. When it receives all confirmations, the update is made visible. Client migrations follow a similar procedure to the remote update dependency check.

4.2.3 Occult

Occult also follows the layout depicted in Figure 4.1. Occult supports three different metadata compression techniques. Given that the experimental deployment considers a single datacenter, the implementation uses the temporal compression of the causal timestamps. Also, the datacenter is the master of all shards, meaning that all client writes are made on its partitions. The size of the causal timestamp is fixed at 10 entries, because, according to the original Occult paper, it provides the highest goodput. Finally, the maximum amount of retries to the client's local replica is set to 3.

For allowing the measurement of the number of retries, Basho needed to be changed. Each operation must return a term that acknowledges its successful completion and the new client state. In addition to these fields, the new version returns the counter that controls the number of attempts. If the value is 0, the request was fulfilled at the first attempt; if the value is 3, the client must go to the datacenter to read a consistent version.

The temporal compression technique keeps the most recent shardstamps that correspond to the latest writes. Both clients and partitions have a merge procedure for combining two causal timestamps into one. Initially, the largest values are on the head of the list, but the new causal timestamp is built by always placing them on the head. This is due to the way Erlang handles list appends, which results in just copying the new entry and not all the previously chosen ones. However, the final structure is ordered in opposite direction, needing to be reversed. The built-in operation to reverse a list is written in C, running outside the Erlang virtual machine. So, the list appends on the head with the final reversal is faster than always appending to the right place.

Because of this being an optimistic scheme, instances can immediately apply the updates to their local state. The FIFO channels ensure that updates are received in the same order as they are created in the datacenter.

4.2.4 Gesto

Gesto follows the layout depicted in Figure 4.2, but with a single broker connecting all instances. The current prototype does not implement the reconfiguration techniques described in Chapter 3.

When processing a write request from a client, the contacted partition starts by changing the clock on the local entry of the client's multipart timestamp (sent on the request) to its current physical clock. Then, the value and the new multipart timestamp are stored in the built-in term storage of the Erlang virtual machine. The object's value is then sent to the sibling partition of other Gesto instances that replicate it and the multipart timestamp is shipped to the timestamp sender.

For applying a remote update, the partition must have both the payload and the metadata. If it receives the data before the metadata, it stores it in a particular table of the term storage. Otherwise,

the multipart timestamp forwarded by the timestamp receiver is stored in a state variable. The use of the term storage allows fast data indexing, because the access is done through a hash function. When both parts are available, the object's value and metadata are stored in the same term storage table as any other update. Finally, the partition acknowledges the timestamp receiver about the completion of the operation, being able to receive more metadata of remote updates.

The timestamp receiver also gives support to the fast client migrations. It keeps the highest local entry from each replica, as well as the highest regional entry ever seen. As explained at Chapter 3, they are used for checking if the local cloudlet state is synchronized with the client's causal history. That is the reason behind clients finishing migrations without having to wait for the faux update.

4.2.5 Saturn

Saturn uses the layout depicted in fig. 4.2. Overall, Saturn shares some behavior with Gesto. However, clients keep even less metadata, migrations follow a different protocol and there is an internal network of brokers for guiding the metadata.

Client reads are very similar to Gesto's. The request contains the same information, but the client keeps the metadata with the highest clock as its dependency. Writes have the same steps, with the new metadata completely replacing the client state. However, migrations have two sequential phases. Firstly, the client contacts its local replica, requesting a move to some target. After acquiring the migration label, the client goes to the target's timestamp receiver and waits until it gets this exact label or a newer one. Clients cannot concurrently perform these phases, because they might store a label associated with an update that was not created in the origin replica.

Regarding the internal network, there is one broker co-located with each replica. In the propagation of metadata, a label may go to a timestamp receiver or to another broker. Eventually, labels reach every replica that stores the associated object.

Summary

This chapter has described a framework for implementing some systems that are used for a fair evaluation of the proposed solution. There are two main layouts: one for systems that propagate the update payload alongside its metadata; other for solutions that differentiate among the two. Although systems may share the same components, their internals have different behaviors, which were also detailed. The next chapter presents a comprehensive evaluation of the proposed solution's prototype.

5

Evaluation

Contents

5.1 Goal 63
5.2 Experimental Settings 63
5.3 Local Operation Latency 65
5.4 Throughput 65
5.5 Remote Update Visibility 66
5.6 Scalability 68
5.7 Migration Latency 69
5.8 Discussion 70

This chapter focuses on the experiments that were done for evaluating the proposed solution. It starts by stating the goal that Gesto must fulfill (Section 5.1). Afterwards, Section 5.2 describes the experimental settings used during the evaluation. The experiments start at Section 5.3, with the analysis of each system's local operation latencies. Then, Section 5.4 investigates the impact of the client access pattern on the system's throughput. Section 5.5 looks on how the client access pattern impacts the remote update visibility latency and discusses some particular situations that differentiate the systems. Section 5.6 checks the evolution of the systems' throughput with the increase of the number of clients. The last experiment is presented on Section 5.7 and addresses some delays that systems introduce in the client migration. Finally, Section 5.8 discusses some systems' bottlenecks.

5.1 Goal

The goal of this evaluation is to determine if Gesto can extend causally consistent cloud stores to efficiently operate at the network edge. Thus, it answers the following question: *can Gesto simultaneously offer low local operation latency, high throughput, fast replication of updates, scalability and fast client migration?*

In order to fairly answer the question, Gesto is compared to three cloud-based solutions on each metric: COPS [2], Occult [14] and Saturn [15]. Furthermore, an eventually consistent solution works as a common baseline, as it provides the best achievable result for each test. It also gives some insights regarding possible overheads.

5.2 Experimental Settings

All experiments were run on the Grid'5000 [47] experimental platform, using fully-dedicated servers. Each server runs Ubuntu 14.04 and has two physical CPUs, ranging from four to eight cores each. Regarding memory, it went from four to 64 GB.

There are seven different locations for deploying the machines. Table 5.1 presents the average measured latencies among these. The single datacenter and the regional broker are on Lyon, because of its centrality. On the remaining locations, there was the deployment of a variable number of cloudlets. Clients are co-located with their preferred replica, but in separate machines. Section 5.6 is the only one that varies the number of client processes. All the others keep two processes per client machine.

Each client machine runs its own instance of a custom version of Basho Bench [45], eagerly sending requests to its preferred replica with zero thinking time. Each experiment runs for two minutes. The first and the last ten seconds of each experiment are ignored, to avoid experimental artifacts. The remote update visibility is computed by storing the physical time at the origin replica when the operation is

	Lu	Ly	Nc	Nt	R	S
Li	5.7	6.6	4.8	13.0	13.6	9.7
Lu	-	8.5	1.2	15.0	15.6	11.7
Ly	-	-	7.5	6.9	7.4	3.5
Nc	-	-	-	14.0	14.6	10.7
Nt	-	-	-	-	0.8	10.0
R	-	-	-	-	-	10.6

Table 5.1: Average latencies (half RTT) in milliseconds among locations: Lille (Li), Luxembourg (Lu), Lyon (Ly), Nancy (Nc), Nantes (Nt), Rennes (R) and Sophia (S).

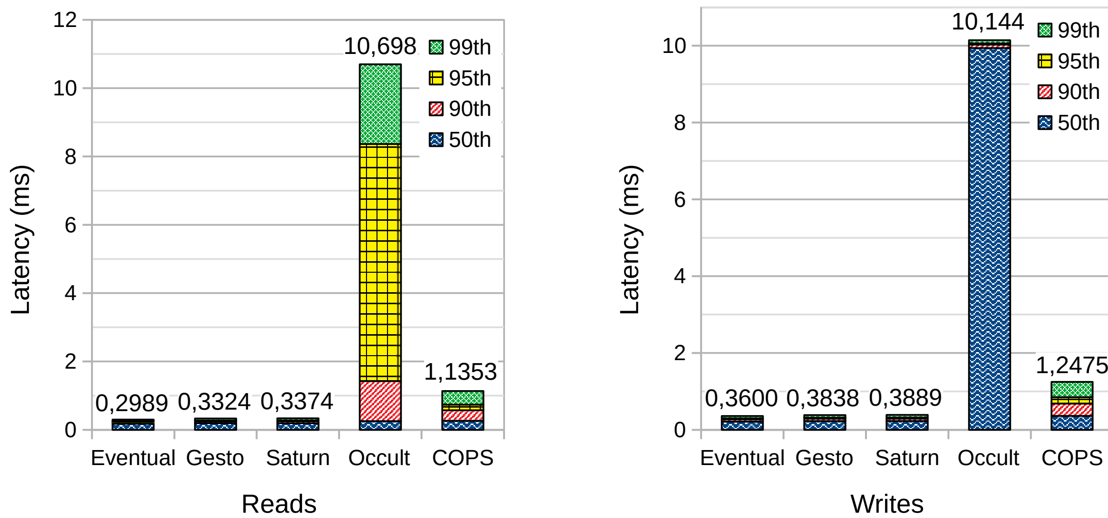

Figure 5.1: Each system's read and write latencies.

issued, and subtracting it from the physical time at the destination replica when the operation completes. To synchronize the clocks, each machine runs the Network Time Protocol (NTP) [48] before starting the test. All the other results are obtained through Basho itself.

Finally, clients can have one of two synthetic workloads: *W1* and *W2*. In W1, clients only perform local operations, with a distribution of 90% reads and 10% writes. In W2, clients execute a mixture of local operations and migrations, with a distribution of 70% reads, 10% writes and 20% migrations to any replica. In both workloads, the client access pattern can be tuned for controlling the number of direct causal dependencies associated with each operation. For instance, forcing a client to read n data items and, only then, to perform the write, approximates the number of dependencies to n. W2 is just considered on Section 5.7, but W1 is used on all experiments.

5.3 Local Operation Latency

The first experiment compares the systems in regard to their client read and write latencies (plots on the left and right of Figure 5.1, respectively). The majority of the systems has faster reads than writes, with Occult being the exception. Its writes take almost the same time (there is marginal difference between the median and the 99th percentile), because they are always made on the datacenter. But, in reads, there is the possibility of the local replica replying with an inconsistent version. When this happens, the client retries two more times and then goes to the datacenter. So, in the worst case, reads take as long as writes, plus the latencies from the retries. In practice, clients only need to visit the datacenter less than ten percent of the times, because the 90th percentile is below two milliseconds.

Looking at the remaining systems, eventual consistency has the combined lowest latencies, for not creating metadata. Saturn and Gesto have virtually the same results, because they support fast metadata creation and the serialization of updates is not on the client path. When compared to eventual consistency, their small delay is mainly due to the creation of metadata. COPS cannot match the performance of these two causally consistent solutions, because of its explicit dependency check. When there is an update to the local state, the server looks for pending dependencies that got fulfilled. This increases the load on the node, making the clients to wait longer for the completion of their operations.

5.4 Throughput

The systems' global throughput of local operations for a given client access pattern is displayed in Figure 5.2. Not surprisingly, the eventually consistent system makes the most operations, mainly due to not managing any metadata. Gesto and Saturn are not far away (they are lower by less than ten percent). Between them, there is no significant difference, because the serialization of remote updates is not on the client path and their propagation is done on the background. Most of the times, clients do not wait for their request to be fulfilled. COPS is the only system that is heavily impacted by the client access pattern. When the client's dependency list increases, the costs involved in explicitly managing causal dependencies are large enough to incur in a system slowdown. At this experiment, Occult exhibits the lowest throughput, because of its high write latency. However, in Section 5.6, there is an increase in the number of clients, showing that it supports a fair amount without being overloaded. Initially, this system had even lower throughput, because the datacenter had to propagate the update to all replicas and the amount of updates that arrived at each instance was the highest. Furthermore, clients write to the datacenter and, when trying to read from their local replica, the probability of getting a non-consistent version is higher, leading to more retries.

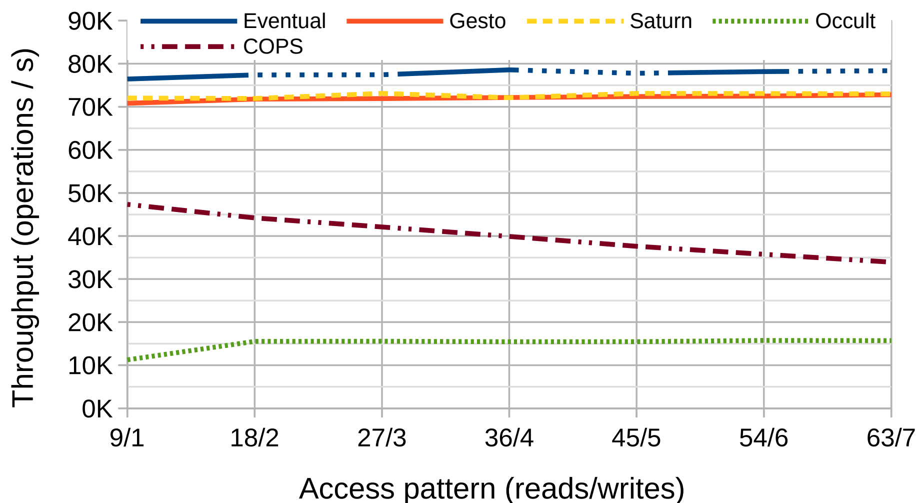

Figure 5.2: Each system's throughput for a given access pattern, while maintaining the same read/write ratio.

5.5 Remote Update Visibility

In Figure 5.3, there are the remote update visibility latencies for each system's access pattern. Like in Section 5.4, only COPS shows a negative evolution, with all the other systems keeping an almost constant remote update visibility latency. Eventual is still the system with the best performance, but now is closely followed by Occult (COPS is near, when clients frequently write to a fully replicated object). Given its optimistic approach to causal consistency, Occult's slave replicas apply remote updates as soon as they receive them, without making any checks. The node organization together with the FIFO channels ensure that updates are received in the same order as they left the master, which preserves the consistency. In the worst case, Gesto shows a smaller remote update visibility latency than Saturn. The more direct propagation of updates and the lower number of hops in the internal nodes are what contribute to such fact.

For visualizing each system's overall remote update visibility latency, Figure 5.4 shows the corresponding Cumulative Distribution Function (CDF). This time, clients read a variable amount of objects between updates, leading to a more generalized behavior, while keeping workload W1. COPS was excluded from this experiment, because there is a more relaxed control of the amount of dependencies and it has high probability of bottlenecking.

The experimental results offer some interesting insights. As expected, eventual consistency is always the fastest, with the results directly matching the latency among the different physical locations. Occult gets closer to eventual consistency as the amount of remote updates increase. Given that all writes are done in the datacenter, geographically close locations incur in a higher remote update latency. This becomes less noticeable as nodes get farther apart, because the detour has a lower impact. Saturn

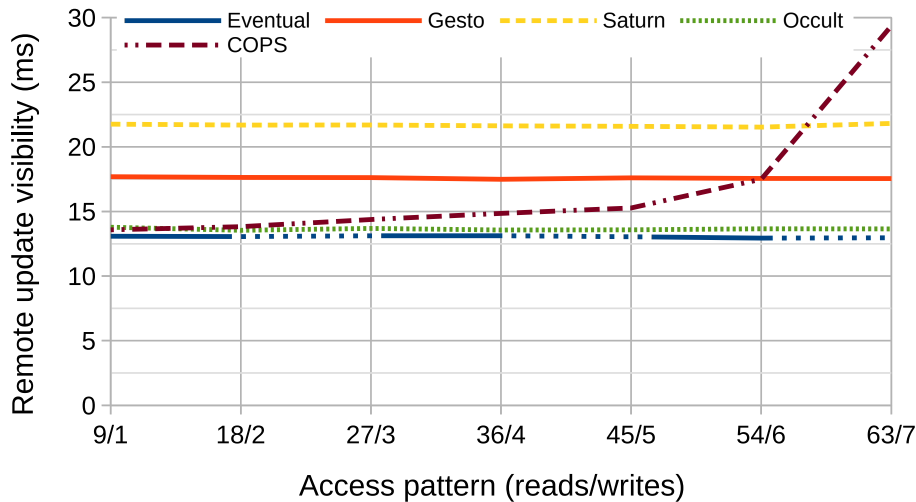

Figure 5.3: Each system's 90th percentile of remote update visibility for a given access pattern, while maintaining the same read/write ratio.

explores the proximity of its instances by having more serializers connecting them. It allows a fast propagation of updates among close replicas, due to updates not travelling as far as the datacenter. However, it quickly slows down (it is only faster on the first five percent of all updates), because the number of nodes the update must go through leads to the introduction of more delays. This is also visible when Gesto surpasses Saturn (around half of all updates).

The comparison between Saturn and Gesto is interesting, because they use different strategies to circumvent the limitation of using small metadata. Saturn is designed for geo-replication, where the latency between replicas varies from a few tens to hundred of milliseconds. This helps the tree topology to be more efficient. By placing the most distant replicas at the extremes of the tree and closely locating sibling replicas, even when the tree inevitably adds some delay on metadata delivery between the most distant replicas, this is relatively small when compared to the latency among those replicas. Nevertheless, within a region, there will not be such significant differences (as shown in Table 5.1), which would make the topology inefficient: for the distant parties (those connected by longer tree-paths), the metadata propagation latency could be several times the latency among the parties. Thus, in Gesto, given the more uniform distribution of latencies among regional replicas and the fact that it is expected for most remote accesses to be performed at the datacenter, the simplicity of the star topology with single broker has competitive results.

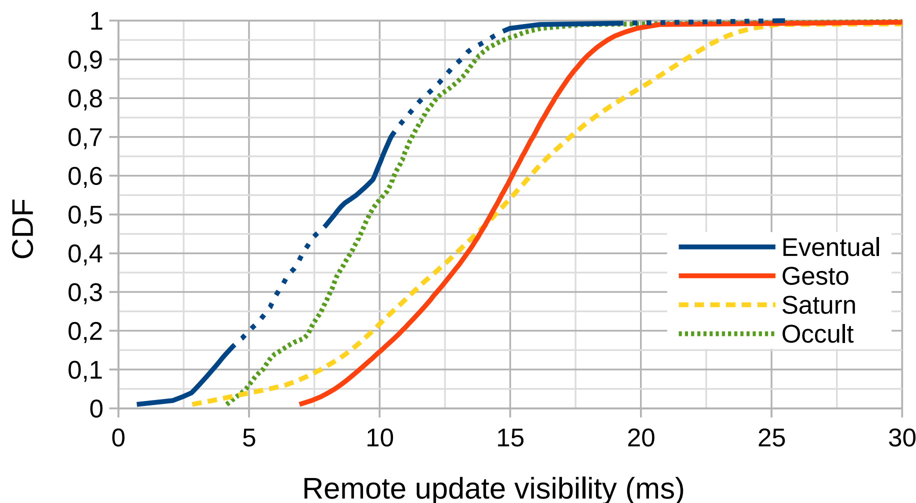

Figure 5.4: Cumulative distribution function for each system's remote update visibility.

5.6 Scalability

This experiment focuses on understanding how many clients are needed to reach each system's maximum throughput. Clients are equally shared among all instances, in an attempt of distributing the load. The results are depicted in Figure 5.5.

Given the different approaches, it is possible to see that the curves evolve in particular manners. Eventual consistency should always have the highest throughput, but this does not happen. From 150 clients, it starts to slow down and is surpassed by both Gesto and Saturn. However, the problem is not with eventual consistency, but with the other two. After a careful analysis, the components for exporting and importing metadata are bottlenecking the propagation of updates (their queue grows faster than what they can process). This results in each machine receiving less remote updates, lowering the load on the partitions and allowing the response to more client requests. An optimization to the code of these components might enhance the remote update visibility and reduce the throughput to values below the eventually consistent system. When looking at a number of clients below 150, the performance of Gesto and Saturn is near optimal, which was also shown in Section 5.4. COPS starts by almost matching the performance of the previous solutions, but, as there are more dependencies to be checked, its performance decreases. At its maximum, the throughput is roughly 45% lower than both Gesto and Saturn. Finally, Occult has an interesting growth. In the beginning, it had the worst performance, due to the high client write latency. With the increase of the load, it showed a steady evolution, getting its maximum throughput when serving 550 clients. It even exceeded COPS after 450 clients. Because Occult relies on simple mechanisms, the servers require less processing time for completing the client requests. Furthermore, the consistency is checked on the client side, which bypasses COPS' biggest

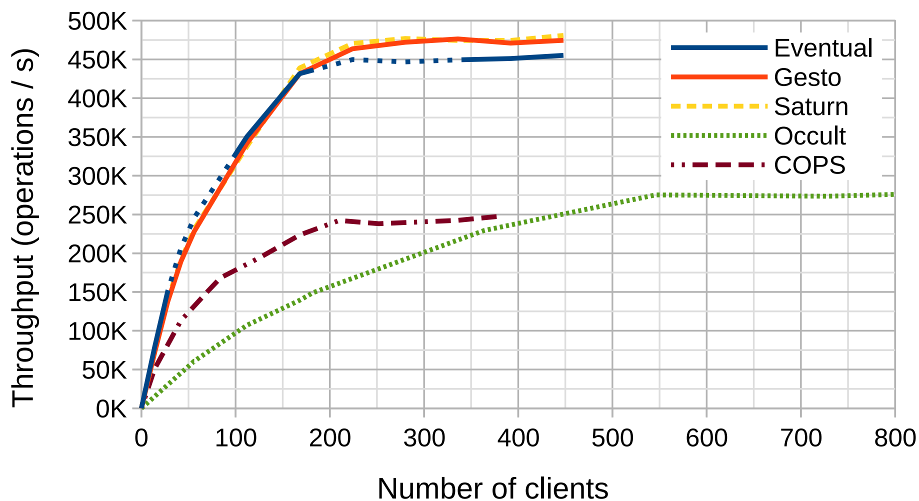

Figure 5.5: Each system's throughput for a given number of clients.

bottleneck.

5.7 Migration Latency

The last experiment is to assess the cost of performing a client migration, mainly the additional latency that is induced by this operation. Each replica has a fixed amount of clients with workload W1 and, for a period of time, can keep clients with workload W2. The CDF with each system's migration latency is displayed in Figure 5.6.

Again, since the eventually consistent system does not enforce any invariant when performing this operation, it is the one that offers faster migrations. Occult, Gesto and COPS exhibit near optimal migration latency. Occult relies on optimistic causal consistency to show the updates as soon as they arrive, reducing the probability of clients having to wait for consistent versions. Gesto's multipart timestamp accounts for client's read-heavy workload. Since clients do not migrate immediately after making a write, the waiting time for the synchronization among the instances is reduced. As the results show, the migration usually finishes without additional latencies. COPS exhibits a little higher latencies, because of the cost associated with the processing of its metadata. Finally, Saturn has the worst latencies. This is due to Saturn not optimizing client migrations, as these are considered very rare. Thus, when a client migrates, Saturn generates a label with a potentially high number of false dependencies, which has an impact on latency. The sequential execution of the label creation and the attachment to the target replica are also responsible for slowing down the process.

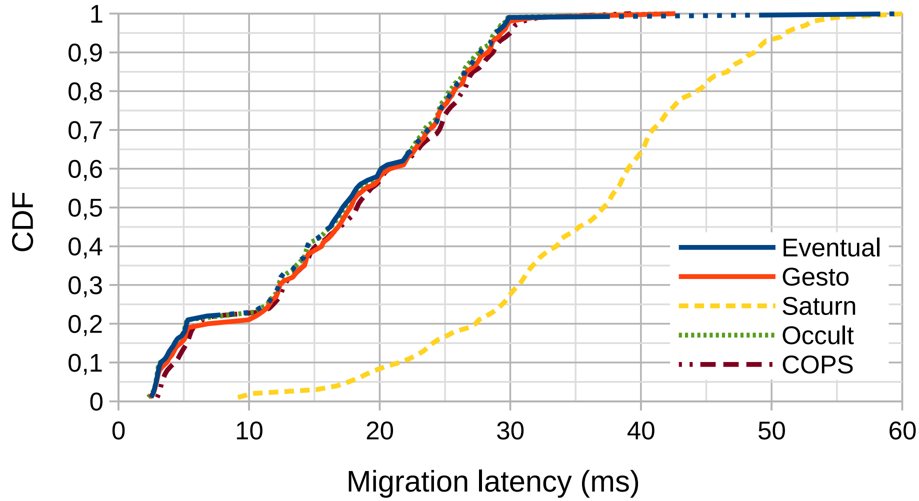

Figure 5.6: Cumulative distribution function for each system's migration latency.

	COPS	Occult	Saturn	Gesto
Fast local operations	±	×	✓	✓
High throughput	×	×	✓	✓
Fast remote update visibility	±	✓	✓	✓
Scalability	×	±	✓	✓
Low migration latency	±	✓	×	✓

Table 5.2: High level comparison of the evaluation results for each causally consistent system.

5.8 Discussion

Gesto is the only system that satisfies all the identified requirements for supporting edge computing. Table 5.2 summarizes the identified differences:

1. Due to its fine grain dependency tracking, COPS has the potential to offer fast local operations, update replication and client migration. This is observed in scenarios where updates carry few dependencies. As the number of dependencies grows, the overhead of processing the metadata cancels these benefits. Contrarily, Gesto relies on lightweight metadata of constant size, independently of the number of dependencies and replicas, which was proved to offer a good trade-off between metadata size and accuracy.

2. Occult excels in low remote update visibility and client migration. Although it is the system that handles more clients to achieve its maximum throughput, the value is still far away from Gesto's. This limitation comes from the requirement of writes being performed on a master shard. On the other hand, Gesto supports a multi-master scheme that enhances scalability and throughput, combined with small and constant metadata.

3. Saturn only fails to support quick client migrations. When a client migrates in Saturn, it has to wait for, at least, all updates that were visible in the origin replica to become visible at the target replica, even when many of these are not in the client's causal past. In Gesto, the multipart timestamp allows a significant reduction on the number of false dependencies, which enables fast client migrations.

Summary

This chapter unveiled the experimental evaluation of Gesto, detailing the tests that were conducted in order to assess its performance. The starting goal has an affirmative answer, because Gesto is able to simultaneously provide fast local operations, high throughput, low remote update latency, great scalability and optimal client migration latencies. Most noticeably, it excels in throughput, scalability and migration latency, while getting competitive results for the other metrics. Overall, the simple network topology with the carefully crafted metadata management shows promising performance as a real-world extension for cloud storage. The next chapter ends this thesis by reporting the most important findings, as well as sharing some ideas for future work.

6

Conclusion

Contents

6.1 Conclusions 73
6.2 System Limitations and Future Work 73

6.1 Conclusions

This thesis has described the design, implementation and evaluation of Gesto, an extension of causally consistent cloud stores to the network edge. By using metadata with small and constant size, it can grow without being limited by the number of replicas or system objects. The propagation of updates separates the content from the metadata, relying on an internal node that joins all replicas. Moreover, clients are free to change their preferred replica at anytime, because they want to get something that is not available or they moved to new place that is closer to another replica.

An experimental evaluation compares Gesto with other three state of the art solutions, as well as an eventually consistent system. When compared with other systems that offer causal consistency, Gesto is the only one capable of offering simultaneously fast local operations, high throughput, fast remote updates, scalability, and low migration latency. Most noticeably, clients are able to migrate to other replicas as fast as in a system offering eventual consistency.

6.2 System Limitations and Future Work

Although the evaluation considered the existence of a single region, Gesto can be integrated with any system that offers causal consistency among multiple datacenters. In the future, a prototype that illustrates this integration should be developed. It will be interesting to study if the overhead induced by the need to convert the metadata used by Gesto into the metadata used natively by the datacenter has any visible impact in the overall performance of an integrated system.

The set of reconfiguration protocols envisioned in the design of Gesto have not been implemented in the current of the prototype. However, they are a fundamental piece to support the deployment of Gesto in a real setting.

Finally, it could be interesting to develop a new prototype in a programming language that does not require the use of a virtual machine. Virtual machines make the deployment easier at the cost of some performance penalty. During the evaluation, it was noticeable that the Erlang virtual machine handles non-fixed structures, such as lists, inefficiently; most of the transformations are too slow, and this alone prevents a system from reaching its maximum throughput.

Bibliography

[1] R. Escriva, A. Dubey, B. Wong, and E. G. Sirer, "Kronos: The Design and Implementation of an Event Ordering Service," in *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*. Amsterdam, The Netherlands: ACM, 2014, pp. 1–14.

[2] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS," in *Proceedings of the Twenty-Third Symposium on Operating Systems Principles - SOSP '11*. Cascais, Portugal: ACM, 2011, pp. 401–416.

[3] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks," in *Proceedings of the Fourth Symposium on Cloud Computing - SoCC '13*. Santa Clara, California, USA: ACM, 2013, pp. 11:1–11:14.

[4] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, "PRACTI Replication," in *Proceedings of the Third Conference on Networked Systems Design and Implementation - NSDI '06*. San Jose, California, USA: USENIX Association, 2006, p. 5.

[5] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing High Availability Using Lazy Replication," *Transactions on Computer Systems*, vol. 10, no. 4, pp. 360–391, 1992.

[6] S. Almeida, J. Leitão, and L. Rodrigues, "ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication," in *Proceedings of the Eighth European Conference on Computer Systems - EuroSys '13*. Prague, Czech Republic: ACM, 2013, pp. 85–98.

[7] 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 Sixteenth Middleware Conference - Middleware '15*. Vancouver, British Columbia, Canada: ACM, 2015, pp. 75–87.

[8] A. v. d. 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 Twenty-Sixth International Conference on World Wide Web - WWW '17*. Perth, Australia: IWWWCSC, 2017, pp. 283–292.

[9] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. M. Preguiça, and M. Shapiro, "Cure: Strong Semantics Meets High Availability and Low Latency," in *Proceeding of the Thirty-Sixth International Conference on Distributed Computing Systems - ICDCS '16*. Nara, Japan: IEEE, 2016, pp. 405–414.

[10] D. Didona, K. Spirovska, and W. Zwaenepoel, "Okapi: Causally Consistent Geo-Replication Made Faster, Cheaper and More Available," *CoRR*, vol. abs/1702.04263, no. February, pp. 1–12, 2017.

[11] C. Gunawardhana, M. Bravo, and L. Rodrigues, "Unobtrusive Deferred Update Stabilization for Efficient Geo-Replication," in *Proceedings of the 2017 USENIX Annual Technical Conference - USENIX ATC '17*. Santa Clara, California, USA: USENIX Association, 2017, pp. 83–95.

[12] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks," in *Proceedings of the Fifth Symposium on Cloud Computing - SoCC '14*. Seattle, Washington, USA: ACM, 2014, pp. 4:1–4:13.

[13] K. Spirovska, D. Didona, and W. Zwaenepoel, "Optimistic Causal Consistency for Geo-Replicated Key-Value Stores," in *Proceeding of the Thirty-Seventh International Conference on Distributed Computing Systems - ICDCS '17*. Atlanta, Georgia, USA: IEEE, 2017, pp. 2626–2629.

[14] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades," in *Proceedings of the Fourteenth Conference on Networked Systems Design and Implementation - NSDI '17*. Boston, Massachusetts, USA: USENIX Association, 2017, pp. 453–468.

[15] 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 - EuroSys '17*. Belgrade, Serbia: ACM, 2017, pp. 111–126.

[16] P. Fouto, J. Leitão, and N. Preguiça, "Consistência Causal em Sistemas Geo-Distribuídos com Replicação Parcial," in *Atas do 10º Simpósio de Informática - INForum '18*. Coimbra, Portugal: FCTUC, 2018, pp. 65–76.

[17] M. Satyanarayanan, "A Brief History of Cloud Offload: A Personal Journey from Odyssey Through Cyber Foraging to Cloudlets," *GetMobile: Mobile Computing and Communications*, vol. 18, no. 4, pp. 19–23, 2015.

[18] C. Streiffer, A. Srivastava, V. Orlikowski, Y. Velasco, V. Martin, N. Raval, A. Machanavajjhala, and L. Cox, "ePrivateEye: To the Edge and Beyond!" in *Proceedings of the Second Symposium on Edge Computing - SEC '17*. San Jose, California, USA: ACM, 2017, pp. 18:1–18:13.

[19] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan, "Cachier: Edge-Caching for Recognition Applications," in *Proceeding of the Thirty-Seventh International Conference on Distributed Computing Systems - ICDCS '17*. Atlanta, Georgia, USA: IEEE, 2017, pp. 276–286.

[20] M. Satyanarayanan, P. Gibbons, L. Mummert, P. Pillai, P. Simoens, and R. Sukthankar, "Cloudlet-based Just-in-Time Indexing of IoT Video," in *Proceedings of the Global Internet of Things Summit - GIoTS '17*. Geneva, Switzerland: IEEE, 2017, pp. 1–8.

[21] G. Ricart, "A City Edge Cloud with its Economic and Technical Considerations," in *Proceedings of the International Conference on Pervasive Computing and Communications Workshops - PerCom '17*. Kona, Hawaii, USA: IEEE, 2017, pp. 599–604.

[22] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The Case for VM-Based Cloudlets in Mobile Computing," *Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[23] M. Patel, Y. Hu, P. Heédeé, J. Joubert, C. Thornton, B. Naughton, J. Ramos, C. Chan, V. Young, S. Tan, D. Lynch, N. Sprecher, T. Musiol, C. Manzanares, U. Rauschenbach, S. Abeta, L. Chen, K. Shimizu, A. Neal, P. Cosimini, A. Pollard, and G. Klas, "Mobile-Edge Computing – Introductory Technical White Paper," ETSI, Tech. Rep., Sep. 2014.

[24] F. Bonomia, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Workshop on Mobile Cloud Computing - MCC '12*. Helsinki, Finland: ACM, 2012, pp. 13–16.

[25] P. Hao, Y. Bai, X. Zhang, and Y. Zhang, "EdgeCourier: An Edge-hosted Personal Service for Low-bandwidth Document Synchronization in Mobile Cloud Storage Services," in *Proceedings of the Second Symposium on Edge Computing - SEC '17*. San Jose, California, USA: ACM, 2017, pp. 7:1–7:14.

[26] E. Ahmed and M. H. Rehmani, "Mobile Edge Computing: Opportunities, solutions, and challenges," *Future Generation Computer Systems*, vol. 70, pp. 59–63, 2017.

[27] S. Mortazavi, M. Salehe, C. Gomes, C. Phillips, and E. de Lara, "Cloudpath: A Multi-tier Cloud Computing Framework," in *Proceedings of the Second Symposium on Edge Computing - SEC '17*. San Jose, California, USA: ACM, 2017, pp. 20:1–20:13.

[28] "Snapchat," https://www.snapchat.com, Accessed: 2018-10-11.

[29] "Snapchat introduces a redesigned app that separates your friends from brands," https://www.theverge.com/2017/11/29/16712704/snapchat-redesign-friend-feed-discover, Accessed: 2018-10-11.

[30] S. Alhabash and M. Ma, "A Tale of Four Platforms: Motivations and Uses of Facebook, Twitter, Instagram, and Snapchat Among College Students?" *Social Media + Society*, vol. 3, no. 1, pp. 1–13, 2017.

[31] J. B. Bayer, N. B. Ellison, S. Y. Schoenebeck, and E. B. Falk, "Sharing the small moments: ephemeral social interaction on Snapchat," *Information, Communication and Society*, vol. 19, no. 7, pp. 956–977, 2016.

[32] J. M. Vaterlaus, K. Barnett, C. Roche, and J. A. Young, ""Snapchat is more personal": An exploratory study on Snapchat behaviors and young adult interpersonal relationships," *Computers in Human Behavior*, vol. 62, pp. 594–601, 2016.

[33] E. Cho, S. A. Myers, and J. Leskovec, "Friendship and Mobility: User Movement in Location-based Social Networks," in *Proceedings of the Seventeenth International Conference on Knowledge Discovery and Data Mining - KDD '11*. San Diego, California, USA: ACM, 2011, pp. 1082–1090.

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

[35] N. Afonso, M. Bravo, and L. Rodrigues, "Armazenamento de Dados com Coerência Causal na Periferia da Rede," in *Atas do 10º Simpósio de Informática - INForum '18*. Coimbra, Portugal: FCTUC, 2018, pp. 29–40.

[36] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[37] Y. Ai, M. Peng, and K. Zhang, "Edge computing technologies for Internet of Things: a primer," *Digital Communications and Networks*, vol. 4, no. 2, pp. 77–86, 2018.

[38] D. Stutzbach and R. Rejaie, "Understanding Churn in Peer-to-peer Networks," in *Proceedings of the Sixth Conference on Internet Measurement - IMC '06*. Rio de Janeiro, Brazil: ACM, 2006, pp. 189–202.

[39] A. Ravindran and A. George, "An Edge Datastore Architecture For Latency-Critical Distributed Machine Vision Applications," in *Proceedings of the Workshop on Hot Topics in Edge Computing - HotEdge '18*. Boston, Massachusetts, USA: USENIX Association, 2018.

[40] E. A. Brewer, "Towards Robust Distributed Systems (Abstract)," in *Proceedings of the Nineteenth Symposium on Principles of Distributed Computing - PoDC '00*. Portland, Oregon, USA: ACM, 2000, p. 7.

[41] 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.

[42] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[43] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *Proceedings of the 2010 USENIX Annual Technical Conference - USENIX ATC '10*. Boston, Massachusetts, USA: USENIX Association, 2010, p. 11.

[44] L. Lamport, "The Part-time Parliament," *Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[45] "Basho Bench," http://github.com/basho/basho_bench, Accessed: 2018-10-12.

[46] "RIAK KV," http://basho.com/products/riak-kv, Accessed: 2018-10-12.

[47] "Grid'5000," https://www.grid5000.fr, Accessed: 2018-10-12.

[48] "NTP: The Network Time Protocol," http://www.ntp.org/, Accessed: 2018-10-12.