Pedro Filipe Veiga Fouto

Bachelor Degree in Computer Science

A novel causally consistent replication protocol with partial geo-replication

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: João Leitão, Assistant Professor, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Co-orientador: Nuno Preguiça, Associate Professor, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Júri

Presidente:	Prof. Hervé Paulino, FCT/UNL
Arguente:	Prof. Miguel Matos, IST/UL
Vogal:	Prof. João Leitão, FCT/UNL

May, 2018

A novel causally consistent replication protocol with partial geo-replication

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Abstract

Distributed storage systems are a fundamental component of large-scale Internet services. To keep up with the increasing expectations of users regarding availability and latency, the design of data storage systems has evolved to achieve these properties, by exploiting techniques such as partial replication, geo-replication and weaker consistency models.

While systems with these characteristics exist, they usually do not provide all these properties or do so in an inefficient manner, not taking full advantage of them. Additionally, weak consistency models, such as eventual consistency, put an excessively high burden on application programmers for writing correct applications, and hence, multiple systems have moved towards providing additional consistency guarantees such as implementing the causal (and causal+) consistency models.

In this thesis we approach the existing challenges in designing a causally consistent replication protocol, with a focus on the use of geo and partial data replication. To this end, we present a novel replication protocol, capable of enriching an existing geo and partially replicated datastore with the causal+ consistency model.

In addition, this thesis also presents a concrete implementation of the proposed protocol over the popular Cassandra datastore system. This implementation is complemented with experimental results obtained in a realistic scenario, in which we compare our proposal with multiple configurations of the Cassandra datastore (without causal consistency guarantees) and with other existing alternatives. The results show that our proposed solution is able to achieve a balanced performance, with low data visibility delays and without significant performance penalties.

Keywords: Distributed datastore systems, causal+ consistency, geo-replication, partial replication.

Resumo

Os sistemas de armazenamento distribuídos são componentes fundamentais em serviços da Internet de grande escala. De forma a satisfazer as cada vez maiores expectativas dos utilizadores em relação à latência e disponibilidade, o desenho destes sistemas tem evoluído na tentativa de melhorar estas propriedades, explorando técnicas como a replicação parcial, geo-replicação e modelos de consistência mais fracos.

Apesar de existirem sistemas com estas características, normalmente não as possuem todas ou fazem-no de forma pouco eficiente, acabando por não as aproveitarem da melhor forma. Para além disso, os modelos de consistência fracos (como a consistência eventual) colocam demasiadas responsabilidades nos programadores para desenvolverem aplicações correctas, pelo que muitos sistemas têm tentado proporcionar garantias de consistência mais fortes, por exemplo implementando o modelo de consistência causal (ou causal+).

Nesta tese abordamos os desafios existentes na construção de protocolos que garantam consistência causal, especialmente na presença de geo-replicação e replicação parcial de dados. Com este fim, apresentamos um novo protocolo de replicação de dados, que permite enriquecer um sistema de armazenamento de dados com estas características com o modelo de consistência causal+.

Adicionalmente, esta tese também apresenta uma implementação concreta do protocolo proposto sobre o sistema de armazenamento de dados Cassandra. Esta implementação é complementada com resultados experimentais obtidos num cenário realista, sendo comparada com várias configurações do sistema de armazenamento de dados Cassandra (sem garantias de consistência causal) e com outras alternativas existentes. Os resultados mostram que a nossa solução consegue um desempenho equilibrado, com atrasos de visibilidade de operações menores e sem penalizações de desempenho significativas.

Palavras-chave: Sistemas de armazenamento distribuídos, consistência causal+, geo-replicação, replicação parcial.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed data stores are fundamental infrastructures in most large-scale Internet services. Most of these services, specially recent ones, demand fast response times [11, 24] since latency can be perceived by users and it has been demonstrated that a slight increase often results in revenue loss for the service [12, 32]. In order to provide low latency to end-users, two important properties of the underlying data store need to be considered: geo-replication and consistency.

Currently, most large-scale services requiring low latency choose to geo-replicate their system. Geo-replication means having system replicas spread across multiple geographic locations, in order to have replicas close to as many users as possible, thus decreasing response times.

This technique however, can be further improved by using partial replication. In a data store supporting partial replication, each replica of the system stores only a subset of the data. By combining partial replication with geo-replication, a service is capable of replicating, in each geographic location, only the data relevant to the users of that location. This is particularly useful for services such as social networks in which the data accessed by users is heavily dependent on their location. Another advantage of using partial replication is the lower resource requirements needed for each replica: while a replica that stores the entire dataset needs an high amount of resources (these replicas are usually materialized in data centers), a partial replica only needs resources proportional to the set of data they replicate. This means that a system using partial replication can use smaller devices as system replicas, for instance set-top boxes, user devices (such as laptops of desktop computers), or even the upcoming 5G network towers. [15, 17, 35].

The other important property of data stores is their consistency model. Consistency models can generally be divided in two types: strong consistency and weak consistency. Strong consistency models are usually used in applications where data consistency is

more important than latency, such as the requirements of applications using traditional (ACID) databases. In services where user-experience is a key factor, weak consistency is the preferred option as these consistency models favor system availability over data consistency.

Being the strongest consistency model that does not compromise availability [3, 28], causal consistency is one of the most attractive weak consistency models, having been implemented in many recent systems [2, 7, 13, 26, 27]. Causal consistency offers some guarantees which are more intuitive for programmers to reason about their applications (when compared to other weak consistency models such as eventual consistency) while enabling high performance and low latency (when compared to strong consistency models).

1.1 Motivation

While many recent systems have implemented causal replication models, they do so using different techniques which result in each implementation having a different behavior. When comparing these behaviors, the main trade-off that can be observed is between data freshness (how long an update takes to be seen by users connected to each replica) and throughput [6, 20]. This trade-off is caused by the way these systems track causality, with some systems trying to reduce the amount of metadata used, which sacrifices data freshness, while others use more metadata, sacrificing throughput (and potentially latency) since more processing time is required to handle that metadata. The data freshness sacrifice is caused by false dependencies as a result of systems compressing metadata. As such, there is not yet a single best way to track (and enforce) causality.

Another challenge that has not yet been solved by modern systems providing causal consistency is partial geo-replication. While there are indeed causally consistent systems supporting partial geo-replication, they do so inefficiently, not taking full advantage of it.

Due to the difficulty of tracking operation dependencies when those dependencies are over objects not replicated in the local replica, these systems require partial replicas to handle metadata associated with items that they do not replicate [4, 26, 36]. This means not only that the metadata overhead will be higher than strictly necessary, but also that data freshness will be sacrificed, as false dependencies are introduced.

Yet another challenge with large-scale replicated systems is scalability. While supporting a small number of replicas can be simple, increasing the number of replicas can introduce overheads that hamper the system's scalability. Such overheads can occur in systems where, for example, the size of metadata is proportional to the number of replicas or where a replica with some sort of central role in the system becomes a bottleneck.

1.2 Problem Statement

The primary goal of this work is explore new designs for replication protocols capable of offering causal consistency. In order to achieve this, we aim at create a solution that can better balance the trade-off between data freshness and throughput (while attempting to maximize both), with support for efficient partial replication and capable of scaling as much as possible without harming the overall system performance.

1.3 Contributions

The main contributions presented in this thesis are the following:

- Simulation work, which consisted in creating a simulator capable of testing different data propagation schemes and the subsequent evaluation of several possible algorithms, in order to find an appropriate one that satisfies the goals of this work.
- A novel protocol for metadata propagation, which guarantees causal consistency between datacenters, while also supporting partial replication in a geo-replicated setting. This algorithm attempts to maximize throughput and data freshness while using very little metadata. Furthermore, it is able to scale with the number of servers materializing the data storage system in each data center.
- A concrete implementation of this algorithm over an existing eventual consistent datastore (Apache Cassandra [24]) and subsequent study of the effects and challenges of enriching an existing data store with causal consistency.

1.4 Document organization

The remainder of this document is organized in the following manner:

Chapter 2 studies related work: in particular this chapter covers the principles and techniques used for data replication; existing consistency models and their characteristics (with a focus on causal consistency); and the most relevant existing systems and which techniques they use to maintain different consistency models.

Chapter 3 discusses the system model assumed for this work, followed by the description of the algorithms proposed, and the reasoning for such proposals.

Chapter 4 describes the simulation work done for studying and validate the algorithms described in Chapter 3.

Chapter 5 shows the implementation of the chosen algorithm and its integration in Apache Cassandra. It also evaluates this implementation and compares it with the base system.

Chapter 6 presents the conclusions and possible directions for future work.

Chapter 2

Related Work

In this chapter we discuss relevant related work considering the goals of the work conducted in this thesis. In particular we focus on the following topics:

In Section 2.1 the various techniques required to implement replication protocols are discussed, we discuss some of the choices that need to be considered for every replication system.

In Section 2.2 we study and compare consistency models, with special interest on causal consistency.

In Section 2.3 causal consistency is studied in more detail. We present the most common ways that existing systems use for causality tracking across operations.

In section 2.4 peer-to-peer networks are discussed, since they are the base of many replicated system.

In Section 2.5 we present a description of relevant existing replicated data storage systems.

2.1 Replication Protocols

Many distributed systems resort to some form of data replication. Replicating data across several replicas is crucial to ensure important properties, such as:

Availability and Fault-tolerance: If a replica (or several) becomes unavailable, due to a crash, network partition, or any other reason, the system remains available since there are other replicas that contain copies of the data to ensure that the system can keep its correct operation. This redundancy of data also makes it very unlikely that any data will ever be lost, even if multiple replicas (even an entire data center) fail at the same time.

Latency: By distributing replicas across different geographic locations (geo-replication), the overall response time of the system can be improved, for users in different (and distant) locations, enhancing user experience.

However, the price to pay for these benefits is the increased system complexity, more specifically there needs to be a protocol to ensure some kind of consistency between all the replicas, and to govern how different clients should access different copies of data in different locations.

In the following discussions, we present some architecture decisions used to characterize replication protocols.

2.1.1 Replica Location

Refers to the physical location of the replicas in a system, which is usually related to the geographic area covered by the system and the importance of latency for users.

Co-located Co-locating replicas consist in having a distributed system replicated on the same physical location. This is usually useful when attempting to minimize the latency between replicas, since having replicas close to each other means they can have direct and fast connection channels to each others. These systems usually serve users geographically close (for instance a web service available only inside a country), since latency could become a problem when trying to serve users globally.

Geo-replicated: Geo-replicating a system implies instantiating replicas in several (usually strategically chosen) geographic locations in order to improve the data distribution. Having replicas physically far from each other may increase latency when synchronizing updates between replicas, however this implies that replicas are closer to the users, thus decreasing latency in communications between the user and a replica (i.e, the service). Since these systems usually focus on user experience, this trade-off is acceptable (and often welcomed).

2.1.2 Replication Schemes

Depending on the service being provided by the system, it can make more sense to replicate all the data to every replica, or instead to only replicate a subset of data in each replica.

Full Replication: In systems that use full replication, every replica is equal having a full copy of all data in the system. Replicas will behave the same way as every other replica: every update is propagated to and applied in every single replica and users can access data from any replica.

Partial Replication: By using partial replication, distributed systems can have replicas that only contain a subset of the system's data. Using this technique can increase the

scalability of replicated systems since updates only need to be applied to a subset of replicas, allowing replicas to handle independent parts of the workload in parallel.

Partial replication is particularly useful when combined with geo-replication, this allows for the deployment of partial replicas that only replicate the data relevant to that geographic location.

Using a social network as an example, there could be a partial replica in Europe that only replicates the data relative to european users. Since european users mostly access data about other european users, that replica would considerably improve those user's experience, while avoiding the need to deploy a full replica (which would incur in higher maintenance costs).

Genuine Partial Replication[19]: Genuine partial replication is a particular case of partial replication, where each replica only receives information (meta-data or the data itself) about data items that it replicates locally. This characteristic makes genuine partial replication systems highly scalable, since they use less bandwidth and replicas need to process less information. However, this is not easy to achieve, since it's much easier to just propagate (meta-)data to every replica and let them decide if that data is relevant to them. Additionally, there is an extra level of complexity in this case related with the handing of more complex operations that manipulate sets of data objects that are not contained within a single replica.

Caching: A cache can be seen as a partial replica, where only read operations are allowed. Caching is often used at the client-side to allow faster response time when reading frequently accessed data and to reduce server load, but can also be used at the server-side to increase the speed for responses to frequent read operations. Caches are usually temporary and are more effective for data that is modified less frequently.

2.1.3 Update Propagation/Synchronization

The method used by a replication protocol to propagate operations is usually dependent on what is more important: offering the client low latency times or sacrificing fast response times in favor of showing the client a more consistent view of data.

Synchronous/Eager Synchronization: Systems implementing eager synchronization protocols usually behave as if there is a single replica of the system state. When a user, connected to a particular server, executes an operation that operation is immediately propagated to and executed in every other replica, and only after this step is a response sent back to the client. These systems are usually associated with stronger consistency models. Since updates are propagated immediately it's easier to maintain a consistent state between all replicas. The cost of executing each update

synchronously means these systems lack scalability since operations take longer to execute for each replica in the system, making eager synchronization protocols too costly for large-scale applications that aim to provide fast user responses.

Asynchronous/Lazy Synchronization: In contrast to eager synchronization, lazy synchronization protocols focus on improving response times. When a user executes an operation, that operation is usually executed on the server immediately and an answer is sent back to the client. The operation is then propagated to the other replicas asynchronously. These protocols often allow for replicas to evolve to divergent states, however they eventually converge as updates are propagated from replica to replica. Lazy synchronization protocols are usually associated with weaker consistency models, since updates are not propagated immediately and replicas are allowed to diverge in state. These system usually scale better than eager synchronization systems, since the synchronization cost is lower. They're also preferred for applications focusing on user experience, due to their lower response times. However, systems using lazy synchronization might have issues when attempting to ensure global invariants over the application state (for example, that a counter never goes below of above given threshold values).

2.1.4 Multimaster / Primary backup

Whether or not there exists a main replica in the system is another common characteristic of replication protocols. While some use a single replica to maintain control of the system, others utilize every replica in a similar way.

Primary-backup: Many classical approaches to replication are based on the primary-backup model, in this model there is one replica (the primary replica) that has complete control over the system, and multiple backup replicas that serve only to replicate the data. Operations are sent only to the primary replica, which executes them and then propagates them (or their results) to the backup replicas, when the primary replica fails one of the backups takes its place.

Systems implementing primary-backup strategies may allow clients to execute read operations in the backups, however the data read from these replicas may be outdated. Implementing strong consistency in systems using this replication model is usually preferred, since it's easy to maintain consistency if only one replica can receive operations. The primary-backup model lacks in scalability, since every operation is executed on a single replica, adding more replicas does not increase the overall performance of the system (in the limit, it may actually decrease it due to the need to spend additional resources of the master to maintain the additional replicas up-to-date).

Multi-master: In a multi-master model, as opposed to the primary-backup model, every replica is equal and can receive and execute every operation, being up to the client to

choose which replica to connect to (usually the geographically closest one). Replicas then propagate their operations in the background and resolve conflicts.

This model allows for better scalability of systems, since increasing the number of replicas can increase the overall performance of the systems. The geographic positioning of replicas can also be used to decrease latency to users, improving user experience. However, since it is hard to implement a strong consistency model in a system where every replica can execute updates without synchronizing, weaker consistency models are usually preferred when dealing with the multi-master model, there are however multi-master systems with strong consistency, resorting to coordination mechanisms, such as Paxos [25], or other coordination systems such as Zookeeper [22].

2.1.5 Multi-version tracking

Some systems that resort to weak consistency models (such as causal consistency) make use of versioning. Versioning is a technique which consists in keeping several versions of the same data item at the same time. The most common uses for versioning are:

- Consistency - In order to maintain consistency in the system, sometimes older versions of data need to be returned to the client (for example when the newer version has not yet been propagated to every replica).

- Transactions - For systems that support transactions, versioning can be useful to allow users to keep operating on the adequate versions of data items that are being accessed and modified by other transactions.

In order to distinguish between data versions each version needs to have some kind of identifier, these identifiers are usually based on whichever technique the system uses for tracking causal relations (for example, a vector clock).

2.2 Consistency models

According to the original CAP theorem [9, 18], it is impossible for a distributed system to provide all of the following guarantees simultaneously:

- Consistency - Showing the user only strongly consistent data

- Availability - Having the system always available to the user, even in the presence of failures

- Partition tolerance - Keeping the system functional and correct in the presence of network partitions

In the context of distributed systems (and particularly in the CAP theorem), the definition of consistency is different from the context of, for example, database systems. Consistency in CAP means that in a distributed system, independently of how the data is stored in servers, users should see that data as if there was only a single up-to-date copy of it.

The CAP theorem further defines that only two out of these three properties can be provided by a distributed system simultaneously however, this formula is misleading [8]. In reality, CAP only prohibits a specific situation: perfect availability and strong consistency in the presence of partitions, which are unavoidable in large scale systems such as geo-replicated systems.

With this in mind, distributed applications usually need to choose between consistency or availability. While it is possible for system to guarantee both consistency and availability in the absence of such failures [8], most systems nowadays are distributed and hence subject to suffer network partitions.

While traditional database systems (with ACID guarantees) choose consistency over availability, recently most systems where user experience is essential to ensure success, as seen in the NoSQL movement for example, choose availability over strong consistency [2, 11, 24].

2.2.1 Strong Consistency

A system that chooses consistency over availability typically focuses on providing guarantees in line with one of the existing strong consistency models. In these models every operation is observed by all users in the same order, meaning that users will always observe consistent states of the system. This kind of consistency is important in situations where always having a consistent, up-to-date state is essential to the overall system correctness. We now discuss two of the most relevant strong consistency models:

Serializability: For a system to provide serializability, every client must see the operations issued to the system in the same order, even if that order does not correspond to the global real-time order in which the operations were actually issued. In order to keep the state of every replica consistent, all replicas must appear to execute operations simultaneously. Without this requirement, a client could read two different values from two distinct replicas.

Linearizability: Linearizability can be seen as a particular case of serializability. In this case all replicas need to execute operations in the same order, however that order needs to be the real-time ordering in which they were issued.

For instance, considering 3 clients *C1*, *C2*, and *C3* issuing 3 operations *op1*, *op2*, and *op3*, respectively and in this order, considering an external unique source of time. To provide serializability the system only needs to make sure every replica executes these operations in the same order (for instance: *op2*, *op3*, *op1*), however to provide

linearizability every replica must execute the operations in the order: *op1*, *op2*, *op3* since that was the real-time ordering in which the operations were issued.

2.2.2 Weak Consistency

Weak consistency models, as opposed to strong consistency models, are typically deployed in scenarios where availability is chosen over consistency. In these models operations may not be seen in the same order by every replica and reads issued by clients may return out-of-date values, we now discuss three consistency models that fall within this category.

Eventual Consistency Eventual consistent systems usually try to achieve high availability. As the name suggests, in this kind of systems, when there are no more updates to a certain data item, all nodes will eventually converge to the same state. This means that, before reaching the converged state, the system may be inconsistent, allowing users to see out-of-date and/or unordered values. To reach a converged state, there needs to be some sort of conflict resolution protocol, with the *last-write-wins* [34] approach being the most common, although the use of CRDTs [33] has gained some popularity recently [1, 23].

Causal Consistency Causal consistency is one of the strongest weak consistency models, being compatible with providing availability in the light of the CAP theorem. This makes causal consistency a very attractive option for systems that need high availability while trying to achieve the strongest possible consistency.

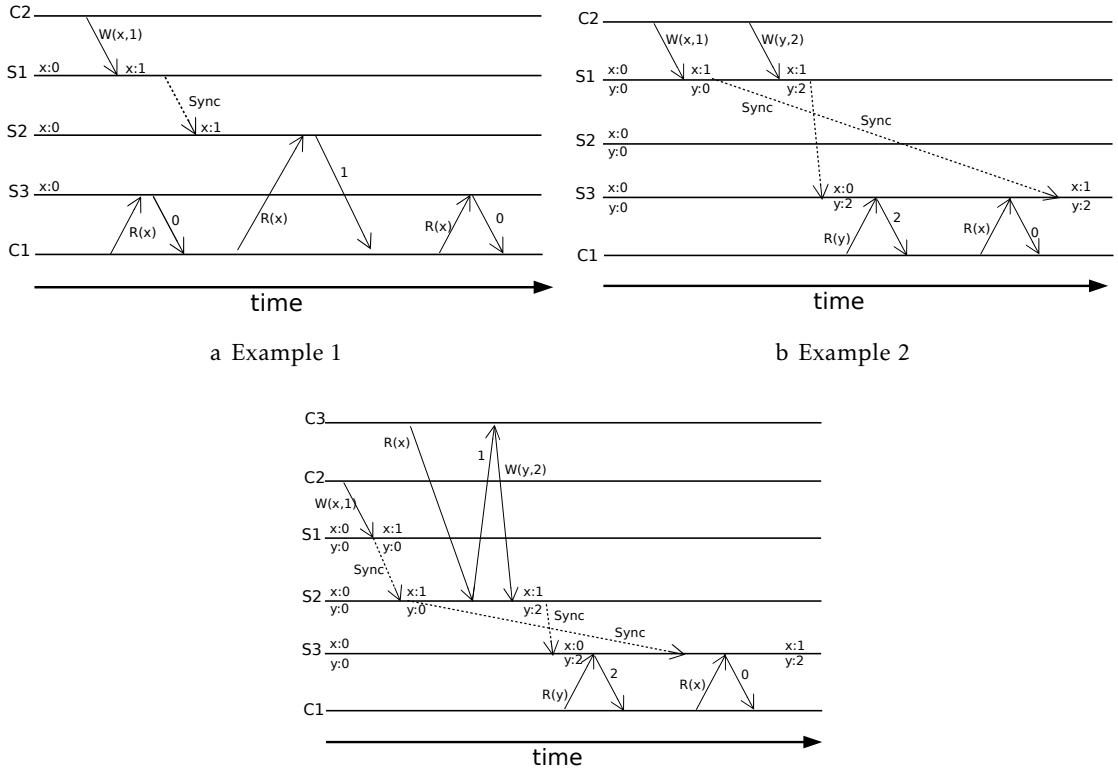
This consistency model requires the system to keep track of causal dependencies between operations and ensures that those operations are always seen by clients in an order that respects their causality relations.

Causally related operation are operations in which one might influence the other, for instance, in a social network, if a user creates a post and then immediately removes it, the remove operation is causally dependent of the create operation.

Operations that do not have any relation between one another, being independent, are called concurrent operations. These operations do not have to be presented to users in any particular order because of this. Simultaneous (and therefore concurrent) writes, for example, are concurrent operations: since they are concurrent, one couldn't influence the other, since it would be impossible for any of them to be triggered by the observation of the effects of the other.

Causal+: Causal+ consistency is achieved by adding convergent conflict handling to causal consistency. Convergent conflict handling ensures that replicas eventually converge, by making them all deal with conflicts in the same way. This property ensures that clients will eventually observe the same (converged) state, if there are no more write operations being performed over the system.

2.2.2.1 Differences between eventual and causal consistency



a Example 1

b Example 2

c Example 3

Figure 2.1: Execution examples that are allowed by eventual consistency but not by causal consistency

In this section, we present examples on the differences between eventual and causal consistency, using Figure 2.1 as reference. In this figure, *c1*, *c2* and *c3* refer to clients operating in a system consisting on three replicas: *s1*, *s2*, *s3*. *x* and *y* represent the keys of objects stored in these replicas. These keys are accessed using read and write operations. These operations are represented by arrows labeled with either and *R* or *W*, where the parameter in a read operation is the key to access and the parameters in a read operation are the key and the new value.

Figure 2.1a shows an example of a situation allowed by eventual consistency but not by causal consistency. In this example, *c1* first reads the initial value of *x* followed by a read to the updated value of *x* and finishing with re-reading the initial value.

In Figure 2.1b, two write operation are issued by *c2*. Since the write operations are issued by the same client, they are considered causally dependent. As they are causally dependent, *c1* should not be able to see the effects of the second one without seeing the effects of the first. As such, this execution is valid under eventual consistency but not under causal consistency.

Figure 2.1c shows a similar situation where *c1* should not be able to read the effects

of the operation issued by *c3* without seeing the effects of the operation issued by *c2*. However, in this case, the causal dependency originates from *c3* reading the effects of the first operation and then issuing a write operation.

2.3 Tracking Causality

Being one of the strongest consistency models for systems that focus on availability, the causal consistency model is a very attractive consistency model. Simply assigning a global order to each operation (using Lamport timestamps, for example) is enough to guarantee causality. However, this is not an efficient method, since concurrent operations will still be ordered without need. A more efficient method to guarantee causality is by tracking causal relations between operations, and then applying those operations in an order that respects these causal relations. This means that only dependent operations will be ordered, while concurrent operations can be executed in any order. Since there is no single best way to track causality, the performance of causally consistent systems is usually dependent on which protocol or technique is used. The basic concept, in which most causality tracking techniques are based, is the concept of causal history.

2.3.1 Causal history

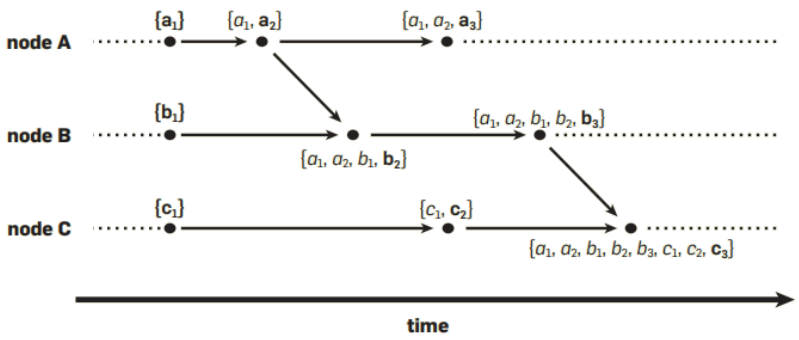


Figure 2.2: An example causal history - Adapted from [5]

Throughout this section we follow the definitions originally presented in [5]. Using causal histories is a very simple way of tracking causality. The causal history of an event can be defined as the set of events that happened before that event. Imagine a system with 3 nodes (A, B, and C) in which, every time an events occurs in a node, that node assigns the event a unique identifier composed by the node name and a local increasing counter. Each event's causal history is composed of its identifier and the causal history of the previous event that occurred at that node.

For example, as seen in Figure 2.2, the second event in node C has the name *c2* and the causal history H_{c2}={*c1*,*c2*}. Also, when a message is propagated to another node, the causal history of the event that is sent is also propagated along. When that event

is received, the remote causal history is merged with the local one. This can be seen in Figure 2.2 when node B receives a message from node A, both causal histories are merged, and a new event b2 is created.

In a system with this behavior, checking for causality is now simple: if an event identifier is contained in another event's causal history, that means the second event is causally dependent on the first; if neither event identifier is contained in the other causal history, then the events are concurrent.

While causal histories do work, the algorithm described above is not easy to implement in an efficient way, as the size of meta-data in a real system implementation would grow infinitely. Several techniques have been created to address this challenge, by using the concept of causal history but in more efficient manners:

Vector clock: By studying the structure of causal histories, there's an important characteristic that can be observed: if a causal history includes an event B3, then it also includes all events from node *B* that precede *b3* (*b2* and *b1*). Given this property, the preceding events do not need to be stored, and only the most recent event from each node is stored.

With this in mind we can, for example, compact the causal history {*a1*, *a2*, *b1*, *b2*, *b3*, *c1*, *c2*, *c3*} into the representation $\{a \mapsto 2, b \mapsto 3, c \mapsto 3\}$ or simply a vector [2, 3, 3]. This vector is called a vector clock.

All the operations performed over a causal history have a corresponding operation identified by a particular vector clock:

- When a new event occurs in a node, instead of creating a new identifier for the event and adding it to the causal history, it's only needed to increase the number corresponding to that node in the vector clock. For instance, after an event occurs in node *B*, the vector clock [1, 2, 3] becomes [1, 3, 3].
- The union of causal histories (when nodes send messages to other nodes), is equivalent to choosing the max value from each position of each vector and placing it in the new vector. For example, the union of the vector clocks [1, 2, 3] and [3, 2, 1] results in the vector clock [3, 2, 3]. Using a more formal explanations, for two vectors V_x and V_y, the result V_z of their union is achieved the following way: $\forall \mathrm{i} : V_z[\mathrm{i}] = \mathbf{max}(V_x[\mathrm{i}], V_y[\mathrm{i}])$
- To check if there's a causal dependency between two events, checking if every position of the vector identifying an event is greater or equal to the corresponding position in the other event vector, and vice versa is enough. Vector V_x is causally dependent on vector V_y if: $\forall \mathrm{i} : V_x[\mathrm{i}] \leq V_y[\mathrm{i}]$ and $\exists \mathrm{j} : V_x[\mathrm{j}] < V_y[\mathrm{j}]$.

Usually, in storage systems, only state changes need to be tracked. As such, a new event identifier only needs to be generated when a write operation occurs (since read operations do not change data). This is called a **version vector**.

Nearest dependencies: Another property that can be observed in causal histories is that an event's causal history contains the causal history of all the events it causally depends on. Going back to Figure 2.2, the causal history of *b2* includes both *a1* and *a2*, however, since the causal history of *a2* already contains *a1*, there's no need to store *a1* in the causal history of *b2*.

This concept is used, for example, by COPS [26]: by only storing the closest dependencies in the causal history of an event it is still possible to transitively rebuild the full causal history of an event.

2.3.2 Metadata Propagation

A different way to guarantee causality is to control the propagation of metadata to ensure updates are executed on remote replicas in a causally consistent fashion. Saturn [7] works by exploiting this observation: it separates data and metadata management, and uses a decentralized metadata manager that delivers the metadata to data centers in a causal order. In order to do this, Saturn organizes every datacenter into a single static tree, in which metadata is propagated through its branches in a FIFO order. Data centers then apply the updates only when they receive the metadata handled by the metadata manager, which guarantees the updates are executed in causal order, even if the data itself is received in a different order.

2.4 Peer-to-Peer

A peer-to-peer system is a decentralized system in which there is no single central server, each peer implements both server and client functionalities. By allocating tasks among peers, bandwidth, computation, and storage are distributed across all participants[31].

For a new node to join the system, there is usually little manual configuration needed. Nodes generally belong to independent individuals who join the system voluntarily, and are not controlled by a single organization.

One of the biggest advantages of peer-to-peer is its organic growth: due to the distribution of tasks, each node that joins increases the available resources in the system, meaning the system can grow almost infinitely.

Another strength of peer-to-peer is its resilience to attacks and failures: since there is usually no single point of failure, it is much harder to attack a peer-to-peer system than it is to attack a client-server system. The heterogeneity of peers also makes the system more resilient to failures since a failure that affects a portion of nodes usually does not affect every node.

Popular peer-to-peer applications include file-sharing, media streaming and volunteer computing.

For a peer-to-peer system to function properly, nodes need to be aware of the underlaying network and its topology. To facilitate communications between nodes, creating

a logical network that only includes the nodes that belong to the system is the most common approach. This logical network is called an overlay network.

2.4.1 Overlay Networks

An overlay network is a logical network, built on top of another network. In an overlay network, nodes are connected by virtual links, which connect two nodes directly through a combination of multiple underlying physical links. In peer-to-peer systems, this underlying network is usually the Internet. The overall efficiency of a peer-to-peer system is dependent on its overlay network, which should have the adequate characteristics to serve that system.

The fundamental choices in an overlay network are the degree of centralization (decentralized vs partially centralized) and the network topology (structured vs unstructured).

Degree of centralization: Overlay networks can be classified by their use of centralized components (or the lack of).

Partially centralized: These networks use dedicated nodes or a central server to have some kind of control over the network, usually indexing the available nodes. These nodes are then used as coordinators, facilitating the entrance of new nodes into the system and coordinating the connection between nodes.

Partially centralized systems are easier to build than decentralized systems, however they come with some of the drawbacks of client-server architectures such as a single point of failure and bottleneck. This bottleneck may also negate the organic growth that characterizes these systems.

Decentralized: In this design, the use of dedicated nodes is avoided, making every node equal. This way bottlenecks and single points of failure are avoided, while the potential for scalability and resilience is higher when compared to partially centralized systems. However, since there is no coordinator node, these network have to rely on flooding protocols to propagate queries/changes, which is less efficient than having a coordinator node. These systems sometimes "promote"nodes to supernodes, these nodes have increased responsibilities are often chosen for having a significative amount of resources. While supernodes may increase system performance (for instance, by helping new nodes enter the system), they may bring some of the drawbacks of partially centralized networks.

Structured vs unstructured: Choosing between structured or unstructured architectures usually depends on the amount of churn[1] the system is expecting to be exposed to and the potential usefulness of key-based routing algorithms to the applications being supported by the overlay network.

[1] Churn is a measure of the amount of nodes joining and leaving the system per unit of time.

Structured overlays: In this kind of overlay network, each node is usually assigned an unique identifier in the range of numerical values that determines the node's position in the network. Identifiers should be chosen at random and the nodes should be distributed in an uniform fashion across the identifier space. This results in the nodes being organized in a structured way, usually named DHT (Distributed Hash Table). This structure works similarly to an hash table: each node is responsible for a set of keys and can easily find the node responsible for any key. Structuring the nodes in a DHT allows for the use of key-based routing algorithms, increasing the efficiency of queries, however, it sacrifices performance when churn is high since the DHT must be updated for each node that enters or leaves the system, which is a process that has non-negligible overhead while also requiring the coordination of multiple nodes.

Unstructured overlays: In unstructured overlays, there is no particular structure linking the nodes, which means that queries are usually propagated by flooding the network. The overlay is formed by establishing arbitrary links between nodes, meaning peers only have a partial view of the network (usually they only know themselves and a few neighbors). In unstructured overlays we have the opposite of structured overlays: queries are less efficient since they need to be propagated to every node to make sure they reach the ones owning relevant content, however this architecture handles churn much better than structured overlays. Since the management of the topology is much more relaxed (i.e has few restrictions).

2.5 Existing systems

In this section, we present the most relevant existing causal consistent systems and give a brief overview on how they track causality.

The following systems are not explained in detail as they do not provide partial replication, which is a key characteristic in the solution we aim at devising. However, they are historically important when considering causal consistency, and they introduced ideas which we leveraged in our work.

COPS[26] was the first system to introduce the concept of causal+ consistency, the strongest consistency model under availability constraints. COPS also contributed with its scalability, being able to track causal dependencies across an entire cluster. It works by checking, for each operation, if its causal dependencies have already been satisfied before making its results visible. It uses client-side metadata to keep track of the dependencies for each client operation.

Eiger[27] is a scalable, geo-replicated storage system that innovates, in relation to COPS, by supporting causal+ consistency using column family data models (popularized

by Cassandra [24]), while most systems support only key-value data models. It also supports both read-only and write-only transactions, even for keys spread across multiple servers.

ChainReaction[2] is a geo-distributed key-value datastore. It uses a variant of the chain-replication technique that provides causal+ consistency using minimal metadata. By using this special variant of chain-replication, ChainReaction is able to leverage the existence of multiple replicas to distribute the load of read requests in a single data center. It also leverages a more compact metadata management scheme, used to enforce causal consistency.

GentleRain[13] is a causally consistent geo-replicated data store. It uses a periodic aggregation protocol to determine whether updates can be made visible or not. It differ from other implementations by not using explicit dependency check messages. It uses scalar timestamps from physical clocks and only keeps a single scalar to track causality which leads to a reduced storage cost and communication overhead, however updates visibility may be delayed.

Kronos[14] is a centralized service, with the purpose of tracking dependencies and providing time ordering to distributed applications. It provides an interface by which applications can create events, establish relationships between events, and query for pre-existing relationships. Internally, in order to keep track of dependencies, Kronos maintains an event dependency graph.

The following systems are explained in more detail as they are the more recent and the closest ones to the solution we wish to implement.

Saturn[7]: Saturn was designed as a metadata service for existing geo-replicated systems. Its purpose is to provide causal consistency to systems that do not yet ensure it by design, in an efficient way. It does this by controlling the propagation of metadata for each update, making sure that it is delivered to data centers in an order that respects causality. For this to work, servers can only apply each update after receiving the corresponding metadata from Saturn, even if that means having to wait after receiving the update data. Saturn also enables genuine partial replication, which ensures its scalability. Internally, Saturn organizes data centers in a tree topology (with data centers as leaves), connecting the tree with FIFO channels. Causality is guaranteed by making sure metadata is propagated in order (using the mentioned channels).

Being a novel approach to causality tracking, we are interested and will use in this work some aspects of Saturn, such as its separation between data and metadata and the handling of metadata by a separate service. We also leverage on the idea of using FIFO channels to guarantee causal consistency.

With this said, Saturn still has some weaknesses that we wish to avoid: organizing the metadata propagation layer in a static tree means that supporting dynamic entry and exit of datacenters in the system is very difficult; using a tree also means that the nodes closer to the root are likely to experience higher load than the nodes closer to the leaves; the lack of any kind of dependency metadata (for instance a vector clock) results in false dependencies.

In addition to these, Saturn shows what we think to be its main fault when we reason about how it could be integrated in a weakly consistent data store: In each datacenter, the order in which remote operations are executed must be the same as the order in which the metadata for these operations was received from the tree structure. This happens because, since there is no dependency information, operations cannot be executed concurrently. In practice, this means that each datacenter can only execute a very small number of remote operations at a time, resulting in very high visibility times and the inability to scale with the number of nodes in a datacenter.

Eunomia[21] is a recent work that provides causal consistency guarantees in a fully replicated geo-distributed scenario. It expands the proposal of Saturn [7] of decoupling the data replication layer from the causality tracking layer. Eunomia allows local client operations to always progress without blocking by relying on a intra-dc stabilization technique. This technique relies on a hybrid physical-logical clock per data partition that is associated with each client operation. Based on these clocks, the Eunomia service gathers progress indicators (either operations or notifications that no operation occurred) from each local data center partition, establishing a total order of operations that is compatible with causality per data center. This total order is then employed by the Eunomia service to execute remote operations on each data center.

Due to the use of a total order, Eunomia can propagate minimal metadata across data centers. However data centers are limited to the execution of one single remote operation per remote data center simultaneously. The use of the local data center stabilization procedure can delay the propagation of operations to remote data centers significantly and hence, affect negatively the global visibility times of operations. Furthermore, and contrary to Saturn, Eunomia cannot support partial replication.

We also present Cassandra that, while not providing causal consistency, will still be used in this work as the base datastore system.

Cassandra[24]: Cassandra is a highly scalable and available, decentralized NoSQL database. It offers eventual consistency of data and allows for some tuning of consistency. For this work we are mostly interested in how Cassandra handles data distribution

and replication, and will ignore inner workings like how the data is stored on disk, indexed, or how clients interact with it.

First, we need to explain some key terms and concepts of Cassandra, which will later be referenced in the document (mostly in Chapter 5):

Node: Represents the basic component of Cassandra, where the data is stored. Usually each node represents a single physical/virtual machine.

Datacenter: Represents a collection of related nodes, which may or may not be part of an actual physical datacenter, but should always be physically close.

Cluster: Contains one or more datacenters. Represent an entire instance of the database.

Keyspace or **partition** is a namespace (or a container) for a subset of data that is usually used to differentiate data according to its role in the application. For instance, there is a default "system" keyspace that is used to store information about the database details and configuration. Keyspaces are essential in supporting partial and geo-replication since each can have a different replication strategy.

Replication Factor: Is the number of nodes of each datacenter that replicate each row of data.

Replication Strategy: Each keyspace must have a replication strategy assigned. A replication strategy specifies in which datacenters the data will be replicated, and the replication factor in each of these datacenters. For instance, the following command creates a keyspace named "users" with a replication factor of 3 in the datacenter "europe" and a replication factor of 2 in the datacenter "canada".

```
1  CREATE KEYSPACE users WITH replication = {
2    'class': 'NetworkTopologyStrategy',
3    'europe':3,
4    'canada':2
5  };
```

Each row inserted in this keyspace will now be stored in 3 nodes of the "europe" datacenter plus 2 nodes of the datacenter "canada", these nodes can be any subset of nodes on these datacenters.

Virtual nodes or **vnodes**: are used to distribute rows across nodes in a datacenter. Each node is assigned a number of vnodes which influences the amount of data that node will be responsible for. This is useful if nodes don't all have the same processing power or, in our case, some nodes are running extra tasks.

Cassandra nodes use Gossip, a peer-to-peer communication protocol to exchange information about themselves and other known nodes. By periodically exchanging

state messages, every node quickly learns about all other nodes in the cluster. This information is used for nodes to know which other nodes are interested in each operation (i.e, which nodes store each data object).

Inside each datacenter, nodes are organized in a DHT (as explained in Section 2.4), which is used in conjunction with the replication factor and vnodes to determine which nodes are responsible for each row of data.

By using different keyspaces with different replication strategies, we can easily setup a partially replicated cluster with geo-replication. We chose Cassandra as the base system for this work both because of this and because of its popularity and well-maintained open source code.

While Cassandra already implements most of the characteristics we desire for this work, it is important to emphasize that is does so while only guaranteeing eventual consistency.

Summary

In this Chapter, we presented all the relevant related work that was studied and used and as basis for this thesis.

In the following Chapter, we will present the system model assumed for this work, followed by the design decisions that were done in order to reach the first iteration of our solution. We also present an algorithm capable of assuring the guarantees we're looking for and explain how it works.

Chapter 3

Algorithms for causal consistency

In this chapter we introduce and discuss some algorithm designs that were initially considered as possible solutions for providing causal consistency in partially replicated databases. The protocol must be scalable with the number of nodes in the system. We are also interested in minimizing the effects of churn and maximizing throughput and data freshness.

We start by presenting the system model assumed for this work. In our design, we separate the system in two layers, the datastore layer and the causality layer. We then present some intuitions about the characteristics of possible algorithms. We finish by presenting and explaining an algorithm capable of ensuring the consistency guarantees that we are looking for.

3.1 System Model

The focus of this thesis is on enforcing causal consistency guarantees under partial replication. We aim at providing such guarantees in the most independent way possible of the concrete datastore. Due to this, we start by defining the minimal set of assumptions that we made regarding the operation of the underlying data storage system.

Storage System Consistency: We assume that the storage system provides eventual consistency across data centers, allowing an update to be executed in a data center without coordinating with other data centers. Updates are propagated asynchronously to other data centers.

Within a data center, we assume that after a write completes, all following reads will return the written value. While a write of an object is in progress, the value returned by a read can be either the old or the new value. Unlike linearizability, it is possible that after a read returns the new value, some other read (issued by other

client) may still return the old value. This is the common behavior of replication systems based in quorums that return the most recently written value (and not the value of the majority). However, we assume that this cannot happen to a single client (i.e., a client cannot read a value and then read an older value). This can be enforced in two ways: by having client caching values returned by read operations; or by simply having clients always contacting the same quorum of nodes when reading a given object.

Partial and Geo-replication: We also assume the underlying datastore already supports (some form of) partial and geo-replication. This means that each datacenter should know which datacenters replicate each data object as to be aware of the datacenters where to propagate updates to that particular data objects.

FIFO channels and variable latency: For now, we assume that it is possible to establish connections between datacenters that produce FIFO (first-in first-out) delivery. This means that messages sent from one datacenter to another are delivered in the same order they were sent. This assumption may not be needed for all algorithms. We also consider the possibility of failures or delays on the communication channels which can lead to variable latency in the delivery of messages between datacenters.

Sharding: Even though we are thinking of datacenters as unique nodes, we're still going to assume they implement some form of sharding. Sharding is a technique that consists in splitting the data that a datacenter is responsible for and storing each piece in a different node of that datacenter. With sharding, a datastore is able to spread the load evenly between all its nodes. For instance, if we execute 100 write operations in a datacenter with 10 nodes, each node will (on average) only need to execute 10 operations, whereas in a datacenter without sharding, a single node would have to execute all 100 operations. This implies that the datastore system can be scaled horizontally within a datacenter.

3.2 Design Considerations

With the specified system model, we now present some intuitions about the principles and techniques we considered in order to design the proposed algorithm.

3.2.1 Layer Separation

As a starting point, we depart from some of the insights introduced by the Saturn[7] system. As Saturn, we decided to separate the system in two layers: the datastore layer and the causality layer. The general principle behind this idea is the following: when the datastore layer receives a write operation from a client, it executes the operation locally and propagates it to the other datacenters that are interested in the operation (i.e, that replicate the data object modified by the client request). Additionally, a label that

represents that operation is also sent to the causality layer. When a datacenter receives an operation from another datacenter, it must wait for the reception of the corresponding label from the causality layer before it can execute the operation.

This separation in layers brings some advantages:

- By using this separation of layers, guaranteeing causality in the system becomes simpler, since we can focus only on the design and properties of the causality layer. By delivering operation labels to the datastore layer in causal order, we achieve causal consistency in the system.
- The (possibly large) operation itself is sent directly to the relevant datacenters in the datastore layer, while the causality layer will only process and propagate smaller labels, which can be relevant to minimize communication overhead at the causality layer.
- This separation will also be useful when implementing the resulting protocol over an existing datastore, since the changes to the datastore code will be minimal. We only need to change the behavior of the datastore in very specific locations (generating a label to send to the causality layer, waiting for the label to execute the operation, and acknowledging executions back to the causality layer).

With this decision made, we moved on to the next stage: how to efficiently organize the causality layer while supporting partial replication, maximizing throughput, minimizing data visibility times, and allowing the presence of dynamic datacenters.

3.2.2 Causality Layer Structure

We started by deciding how the causality layer should be organized. Saturn organizes all datacenters in a single tree topology. Having all datacenters organized in this fashion means we would not need to use any extra metadata (like vector clocks or explicit dependencies) to track causality, we would simply need to send all labels through the tree (assuming FIFO channels).

Instead of having the datacenters organized in a single tree structure, we can also propagate labels using different (and independent) trees depending for example, on the datacenter where the operation originated. This translates in having one tree per datacenter in the system, where each datacenter is the root of its own tree. These trees could then be optimized depending on the distribution of data in the system (i.e, the partial replication scheme employed), in order to better distribute the load in the causality layer. Having multiple trees, however, would mean that sending labels in a FIFO channel is no longer enough to guarantee causality, since causality is only guaranteed between operations propagated through the same tree. As such, in each label, we need to append a vector clock. Since each tree by itself guarantees causality between operations that transverse it, this clock effectively tracks dependencies between trees, which means the

size of the clock depends on the number of trees (i.e, the number of datacenters). While this addition of a vector clock to each label will increase the load on the causality layer, some metadata will always be required, as we will explain further below in this section.

A particular case of this multiple tree layout would be having trees where the root connects to every other (relevant) node, which would result in every label being sent directly to every interested datacenter. In this case, supporting high churn is trivial, since we don't need to change anything in the layout of the causality layer when a datacenter enters or leaves the system. On the other hand, with either single or multiple trees, a datacenter joining or leaving requires the trees topology to be reconfigured, which would require that a part of (or even the entire) system stops executing operations while this reconfiguration takes place.

3.2.3 Concurrency and False Dependencies

Having the causality layer organized in a single tree may seem like a good idea at first, since, as explained before, the tree by itself guarantees causality. The most significative issue with this approach is the lack of concurrency in the execution of remote operations. Since datacenters are receiving each label in a specific order, without any information about their dependencies, they have to assume that every label is causally dependent of all labels previously received, which means all operations must be executed one by one in the order they were received. This effect is called "false dependencies" and, as explained in Section 1.1, it is usually mitigated by attaching more precise dependency information to each label. Since we are considering that the datastore layer supports sharding, having no concurrency in the execution of remote operations becomes even more penalizing, since the advantages of sharding is lost (for remote operations). As a result of these observations, we conclude that labels must have some dependency information attached to them and, as such, it makes more sense to rely on multiple trees over a single tree.

3.2.4 Vector Clock vs Explicit Dependencies

After having decided that we will be using multiple trees and that we need dependency information in labels in order to support concurrent execution of operations, making it possible to take advantage of sharding in the datastore layer, we're left with another decision: how fine-grained should that dependency information be.

Here we consider two options: maintaining a vector clock with one entry per datacenter and attach it to the labels or, using a more complex option, having the clients keep track of direct dependencies explicitly and attaching them to every operation.

By only using a vector clock, the effect of false dependencies is more visible, when compared to using explicit dependencies. However, this solution has the advantages of adding very little extra metadata to labels, which is important in order to keep the causality layer from being a bottleneck to the system, while also respecting the separation between layers, by avoiding to require the storage of explicit dependencies in the

datastore layer.

On the other hand, keeping track of explicit dependencies can increase the throughput of the system by allowing the execution of more operations concurrently, which also results in taking more advantage of sharding. However, such solution is much more complex to implement since it requires a client library to be running between the client and the datastore layer, changing the datastore in order to receive the client dependencies and send new dependencies back with answers for client operations, and requiring the datastore layer to share information with the causality layer (to know which dependencies are missing), which may not be trivial to achieve. The overhead of having to handle explicit dependencies in the causality layer may also decrease the overall throughput of the system.

In the next section, we present an algorithm which uses the previous techniques and is able to achieve the goals of this work: a protocol capable of providing causal consistency to partial and geo-replicated datastore.

3.3 Algorithm Design

Based on the discussion presented in the previous section, we now present our thought process when creating the first iteration of our final algorithm. We start by presenting some ideas that were thought of but eventually abandoned, followed by the chosen algorithm.

All the discussion in this section is based on the materials presented in the previous section and considering the system model presented in Section 3.1.

At first, we thought about designing the algorithm using fine-grained dependencies. This means having the client to store information about the data it interacted with, as explained in the previous section. The main problem with this solution is that we would have to propagate that dependency information across the causality layer, which could significantly increase the load on it. We concluded that, in order to avoid the causality layer to become a bottleneck, which would have a significant negative effect on the visibility times in the system, especially when the datastore layer can be horizontally scaled, we need to avoid the use of explicit dependencies. In addition to this, we are also attempting to separate the causality layer as much as possible from the datastore layer, which requires us to avoid having causality tracking information in the datastore layer (and consequently in the client).

Having discarded this option, we then decided to approach the problem in a different way by identifying the minimum amount of metadata to maintain causality while still having enough information to allow some concurrency in the execution of operations. As explained in the previous section, when using multiple trees, the causality information we require is a simple vector clock with one position per tree. This vector clock is also enough to have some amount of concurrency, since a datacenter can simultaneously execute one operation per tree (instead of a single operation at each time when using a single tree).

Using this option also means that we should be able to keep all the dependency tracking information on the causality layer, thus keeping the datastore layer mostly unchanged.

We then decided which logic we are going to use to decompose the message propagation across trees. Our first thought was to use a tree for each data partition (the basic unit for controlling the partial replication scheme). With this structure, each datacenter would only be part of the trees corresponding to its data partitions, which means we would have genuine partial replication. Since we need a vector clock with a position for each tree, as explained before, in this case each datacenter would keep a clock position per each replicated partition (without needing to keep clock position for non-replicated partitions). The problem with this approach is that each datacenter, when executing an operation, would increment its clock positions independently which would make it hard to track the correct dependencies between operations executed in different datacenters. This happens because there would not be a single datacenter responsible for each clock position. A possible solution for this problem would be to have a main datacenter for each partition, which would be responsible for ordering every write operation in that partition with the correct clock value. However, we did not want to use this solution since it would mean the final system would not be fully decentralized, thus losing availability. Another solution for this problem would be to decompose each clock position into multiple positions (one per datacenter). With this solution, each clock position would again be incremented by a single datacenter. While we believe this could be a good solution, we decided not to use it due to the increased clock size, as this would counter our goal of minimizing as much as possible the load on the causality layer.

In the end, we decided that having a tree for each datacenter would be a better solution than having a tree per each partition. With a tree per datacenter, the size of the needed vector clock would amount to one scalar per datacenter in the system. This way we minimize the load in the causality layer, in order to prevent it from becoming a bottleneck for the datastore layer.

3.3.1 Proposed algorithm

With our thought process explained, we now present what our proposed algorithm looks like. Since we are abstracting the datastore layer, this explanation focus on the causality layer. In Section 5 we explain the actual implementation and integration of both layers.

As stated before, each datacenter has its own propagation tree, and uses it to propagate metadata regarding write operations received from clients at their datacenter. This tree can have any topology, in Chapter 4 we study the effects of the different tree topologies in the visibility times for remote operations. The metadata used to track dependencies consists only on a vector clock with one entry per datacenter. This implies that causality tracking is completely transparent to both the client and the datastore layer. Having this vector clock also means we have enough information to promote concurrency, enabling the execution of remote operations in parallel.

We designed the algorithm in order to support both remote operations and migrations. Remote operations and migrations are the two possible ways of a client executing operations over data that is not replicated at its local datacenter. When using remote operations, the client simply issues its operations and the system is then responsible for routing the client operation to a datacenter that replicates that data. This datacenter will then execute the operation and reply to the client. Since we are building a causal system, we need to make sure that remote operations are propagated in a order that respects causality relations among operations, we leverage on the causality layer to do this. When using migrations, the client issues a special migration request to the system. This message is then propagated, in a causal manner, to the target datacenter (through the causality layer). When this message reaches the target datacenter, the client is notified and can start executing operations normally on its new datacenter. In both of these scenarios, the enforcing of causality is transparent to the client, as it simply issues operations and waits for the response of the system. In Chapter 4 we study the differences between remote operations and migrations from a more practical stand point.

We now explain how each type of operation (local read, remote read, local write, remote write and migrate) is handled by our algorithm:

Local Write: For executing a local write, the client issues the operation to the (local) datastore layer which generates a label representing that operation and handles it to the causality layer, while also propagating the operation directly to the other relevant datacenters. The causality layer then increments its own position on its local vector clock and tags the label with the new clock. It then delivers the label to the local datastore layer, allowing it to execute the operation and also propagates it across the causality layer to every other datacenter using the originator datacenter tree. Once the operation is executed in the local datacenter, the client is notified that the operation has completed. When each remote datacenter receives the label, it waits until its local clock is in a state that allows it to be executed (i.e, until all local clock entries are greater or equal to the operation clock's entries). When this happens, the clock position for the operation's original datacenter is increased and, in case the data relative to the operation is replicated locally, the label is delivered to the datastore layer, thus enabling the execution of the operation.

Remote Write: Remote writes are similar to local writes, with the difference that the operation must first be causally propagated to a datacenter that replicates data, before being executed. When the client executes a remote write, the datastore layer, similar to a local write, propagates the operation to all relevant datacenters and generates a label for the causality layer. The causality layer, however, does not increase the clock, since the operation will not be executed locally. Instead it just tags the local clock to the label and propagates it through its tree to the target datacenter that will execute the operation. When the target datacenter receives the label, it waits until its all its local clocks entries are greater or equal to the label's

clock entries. When this happens, it executes the write operation as if it was a local one, including increasing its local clock, attaching it to a new label and propagating it to the other datacenters.

Local Read: This is the simplest operation since it doesn't require the use of the causality layer. The client simply issues the read operation to the datastore layer and receives the response back immediately.

Remote Reads: Remote reads are more complex than local reads, since they require the use of the causality layer. When a client issues a remote read operation, the datastore layer generates a label for that read and propagates the operation to the remote datacenter. The causality layer then attaches (without increasing) its local clock to the label and propagates it through the tree to the remote datacenter. When the label reaches the remote datacenter, it is delivered (when possible) to the datastore layer, which executes the read. The remote datacenter then needs to send the response back to the client's datacenter through the causality layer. This is required to make sure that subsequent operations made by the client are causally consistent. As such, it sends the response directly through the datastore layer, but also generates a new label to which the causality layer attaches its clock and propagates to the client's datacenter. When the response label arrives at the client's datacenter and can be executed, the client receives the response.

Migrate: A migrate operation is somewhat simpler than remote operations. The client issues the migrate operation to the datastore, which then generates the label and propagates it to the causality layer (the client could also create the label and send it directly to the causality layer). The causality layer then attaches the local clock to the label and propagates it to the target datacenter. When the label reaches the target datacenter it waits until its local clock matches the one in the label, at which point it replies to the client. At this point the client can start issuing local operations on its new datacenter.

Summary

In this chapter we presented the design decisions we were faced with when designing the first iteration of our solution. We also presented the achieved solution, detailing how it works and how it enforces causality. In the next chapter, we present some simulation work used to test and compare some variants of this algorithm.

Chapter 4

Simulation Work

In this chapter, we present a study, done by simulation, of the implications of design decisions and possible alteration for our algorithm.

In order to do this simulation work, we created a simple simulator in which we implemented the causality layer's propagation variants we wished to test. The main purpose of this simulator is to try to understand the effects of different tree topologies on the visibility times of operations. We also use this simulator to understand whether using remote operations is a better option over migrating clients between datacenters.

4.1 Model

The simulator was build considering the system model presented before, albeit attempting to fully abstract the datacenter layer focusing primarily on the causality layer. As such, we assume each datacenter to be composed by two simulator nodes which capture the causality and the datastore layers, respectively. Since we are abstracting the datastore layer, we are not going to simulate the effects of the algorithms when this layer is scaled horizontally (i.e, partitioned), which (as we will see in the next chapter) can have an important effect on the visibility times of remote operations.

4.2 Architecture

The architecture of the simulator is divided in two main entities: clients and nodes. These entities communicate by sending messages to each other.

Client: The client is used to simulate an user executing operations in the system. Its behavior is simple: it generates an operation and sends it to its local nodes, then remains idle until a response is received, at which point it generates a new operation

and sends it. This cycle is repeated until a specific number of operations have been completed. The client can be configured to control its generation of operations, such as the distribution between local and remote operations, the total number of operations to generate, the use of remote operations or migrations, etc. When executing the simulations, the number of clients can be specified, in order to better simulate the normal utilization of the system.

Nodes: The nodes are used to represent the components of the datastore system. Each node can represent either the datastore or the causality component of a datacenter. Its behavior and state depends on the implementation used when running the simulation and can be completely arbitrary. The simulator code supports nodes sending messages to each other or to clients, simulating the latency between them. It can also simulate the processing time of requests in each node.

4.3 Implementation

As explained in the previous section, the clients are implemented in the simulator and behave by synchronously sending operations to nodes. The simulator uses a configuration file to parameterize each simulation. This file has information about the number of clients to use, the number of operations per client, the percentage of reads and writes, the distribution of local and remote operations, and which causality tracking metadata propagation scheme to use in datastore nodes.

The simulation starts by executing the code responsible for creating the nodes. This code is specific for each protocol and is also used to initialize the state of the simulated system and informing the simulator of the connections and latency times between nodes.

Each client is then assigned to one of the created nodes, which will be used as the local node for that client. The latency between a node and a client is the same as the latency between that node and the client's local node.

The simulation itself is event driven, with a sorted queue containing all events generated by nodes and clients in the simulator. Events can be of two types: message *propagation* and message *processing*. When the simulation starts, each client generates its first operation and sends it to a node. This behavior generates a message propagation event for each message sent, which is added to the queue. The time assigned to each of the events is the arrival time at the target node. The simulator then advances time to the next event in the queue.

Since the first event is always a client message reaching a node, the simulator then delivers the message to that node, which will execute the code specific to the metadata propagations protocol being simulated, and return a number representing the amount of time it will take to process that message. The simulator then generates a *processing* event for that node which will be used to signal the end of the message processing.

Time is then advanced to the next event in the queue. If the next event is also a *propagation* event, the same sequence happens. If it is a *processing* event, then the last message received by the node related to the event finishes being processed and, if the behavior of the node includes sending new messages, they are added to the event queue. If the node had received more messages while processing the previous one, then it immediately starts processing the next message (generating a *processing* event), otherwise it stays idle until a new message is received. When a client receives a message from a node (which represents a response to its latest operation), it simply generates a new operation and adds the associated propagation event to the event queue.

The simulator then repeats this behavior until no more events are in the event queue, which means all clients have executed all operations and the simulation has finished.

After the simulation ends, results about client latency, execution times, and data visibility times are saved to disk, in order to be later analyzed (by automated tools created for this purpose).

4.3.1 Protocol Implementation

In order to better understand the implications of different design choices for building metadata dissemination schemes, we started by implementing two different causality protocols: one based on Saturn's propagation scheme, which uses a single tree with no additional metadata besides the operation identification (i.e, a scalar) and another based on the algorithm described on the end of Chapter 3, which uses multiple trees and a vector clock to keep track of causal dependencies. We also implemented some variants of the latter protocol, which differ on the topology of the used tree. We detail these further ahead.

4.4 Experimental Evaluation

In this section, we present the simulation configuration used in our experiments, followed by the analysis of the results obtained.

4.4.1 Configuration

In order to run the simulations we first needed to choose the distribution of nodes and partitions to simulate. We decided to simulate 6 datacenters consisting of *Amazon Web Services (AWS)*[1] datacenters, spread across the world in the following locations: East United States, West United States, Europe, Brazil, Japan and Australia. After choosing these locations, we gathered information about the latency between each pair of datacenters from *AWS*, whose results are summarized in Table 4.1.

We also distributed the data stored across 18 partitions and attributed these partitions to different datacenters, with the distribution shown in Table 4.2. To achieve this, we

[1] https://aws.amazon.com/

	East US	Europe	West US	Brazil	Australia	Japan
East Us	-	50	32	70	112	132
Europe	50	-	71	104	170	133
West US	32	71	-	99	97	56
Brazil	70	104	99	-	195	159
Australia	112	170	97	195	-	71
Japan	132	133	56	159	71	-

Table 4.1: Latencies between simulated datacenters (ms)

started by assigning a main datacenter to each data partition. We then replicated each of these partitions to one or two closest datacenters, and additionally to one or two random datacenters. Each partition is assigned to a number of different datacenters up to 4.

Partition	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
East US	x			x			x			x	x		x			x	x	
Europe		x						x	x	x			x	x	x	x		x
West US			x					x	x			x		x	x	x	x	x
Brazil	x			x			x	x		x			x	x		x		
Australia					x		x				x	x	x		x		x	x
Japan			x		x	x			x		x	x		x	x		x	x

Table 4.2: Partition distribution across datacenters for the simulation experiments

Since the main purpose of the simulation is to determine the best way to materialize the causality layer, we created several different tree topologies to evaluate in the context of the algorithm presented in Chapter 3

Ring Topology: This topology was reached by first finding the shortest ring that spanned all datacenters. Each datacenter then uses that ring as its propagation tree by propagating messages in both directions (without them crossing on the other side of the ring). This results in trees with a large height and where the root has two child nodes and every other node only has one (or none in the leaf nodes).

Ring Core (or Semi Ring) Topology: This topology is based on the previous one however, the two datacenters with the highest average latency to every other were removed from the ring and connected to the closest datacenters in the ring. This results in a ring composed with the four northern hemisphere datacenters, with the two remaining datacenters (southern hemisphere) as "branches" on the ring. Each datacenter uses the tree rooted in itself to propagate its messages to every other datacenter.

Clique Topology: In this topology, each datacenter simply connects to every other datacenter and propagates its messages directly to each one of the remote datacenters. It can be thought of as a tree per datacenter with an height of two, where the root connects directly to every other node.

To emulate the behavior of the Saturn system, we also implemented a solution that relies on a global tree to propagate metadata.

As for the client configuration, we executed the simulations using 600 clients, distributed equally across all six datacenters. Each client issues 3000 operations, half of which are write operations and the other half read operations. Regarding data partitions, clients alternate between executing operations over a partition replicated on the local datacenter and a partition not locally replicated. The percentage of operations over local partitions is 95%, while remote partitions is 5%. When executing operations over remote partitions, we simulate two different client behaviors: sending remote operations through the local datacenter, or migrating to a datacenter that replicates the data to be accessed and executing local operations.

Since we also want to study the effects of having additional causality information in the causality layer, we also run simulations considering different data object sizes. This parameters represents the average size of data written to the system in a write operations or read from the system in a read operation.

To better understand the effects of different causality layer organizations, and since we are abstracting the datastore layer, we assume the datastore layer to have infinite resources, allowing it to execute operations very quickly. We do this to avoid making the datastore layer a bottleneck for the propagation of labels by the causality layer, which would result in every tested topology and strategy to yield similar results.

4.4.2 Results

We now present the results obtained with the configurations described above.

4.4.2.1 Data Visibility

Figure 4.1 shows the visibility times of write operations when using the different propagation topologies, also comparing the use of remote writes against migrating between datacenters. When comparing remote writes and migrations, the results are pretty much the same, with a very small advantage when using migrations. This happens since the propagation of write operations from the datacenter where the operation originated to every other datacenter is independent from whether the client using remote writes or migrations. The small difference represents the time difference between the client executing an operation in its local datacenter or in a remote datacenter.

When comparing the different topologies, differences are more noticeable. Since the ring topology uses a tree with a high height value, writes take more time to be propagated to every datacenter, resulting in the worst visibility times. In the global tree topology, the high visibility times are due to the use of a single tree, which makes it hard to optimize the propagation path for every datacenter. The writes often need to be propagated all the way from a leaf node (i.e, datacenter) to the root node, and then back down to another leaf. When using the semi-ring topology, since each datacenter can choose its own optimized

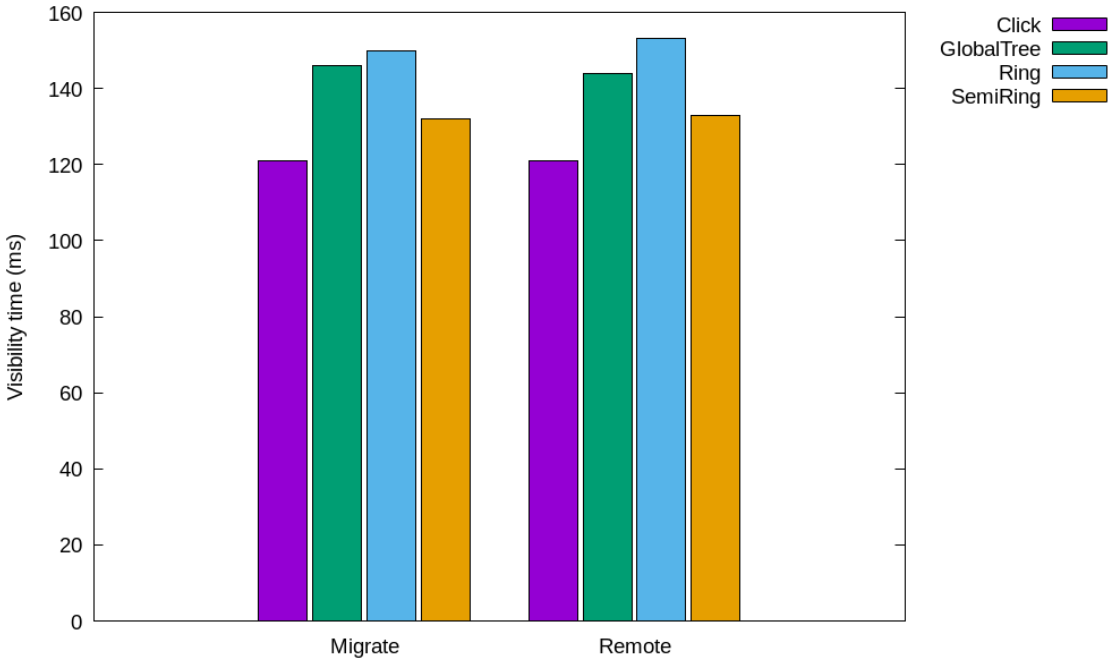


Figure 4.1: Operation Visibility Times

propagation tree, results are better when compared with the use of a global tree. The best results are achieved with the use of the clique topology, since this configuration allows the metadata to be propagated directly to the target nodes without having to go through multiple hops. Due to this, visibility times are always minimal, which implies that this topology is the best when considering the visibility times for write operations.

4.4.2.2 Message Size

Figure 4.2 represents an approximation (since the real value depends on the method of serializing messages) of the number of bytes (considering both operations and metadata) propagated by the datacenters using each topology, with the clients using either remote operations or migrations, and considering different sizes for the data objects being accessed in the datastore.

When comparing the different topologies we can see that the global tree solution, due to its use of less causality tracking information, needs to propagate less data across datacenters. This effect is particularly visible when the size of the data queried and written by clients is very small as seen in the leftmost group of results. However, as the size of that data increases, the difference between the various alternatives becomes less relevant as the size of data objects dominates the results.

We can also observe a small difference between clients executing remote operations or using migrations. This difference is due to remote operations needing to cross multiple datacenters through the tree used to propagate causality tracking metadata, whereas using migrations allows the client to communicates directly with the datacenter that

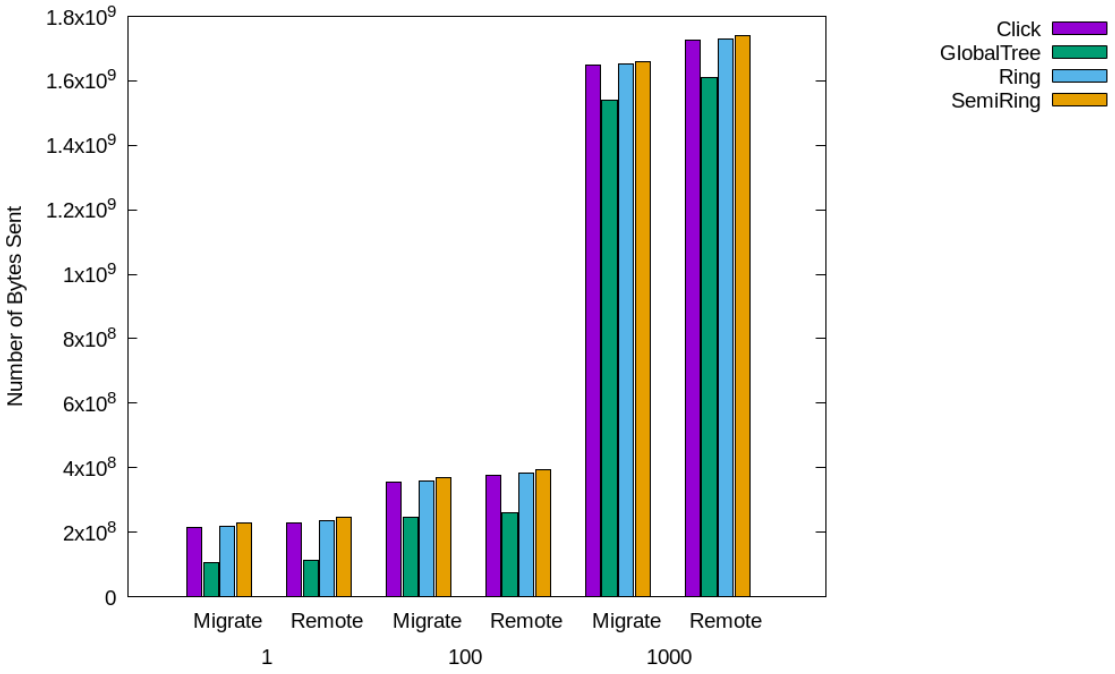


Figure 4.2: Data Propagated Size

replicates the data over which the operations will be executed. This difference is so small because most operations are executed over local replicas and remote operations are significantly less frequent (only 5%).

4.4.2.3 Throughput

Figure 4.3 reports the throughput of the system when using each alternative tree topology, differentiating between the use of migration or remote operations when accessing data not accessible in the local datacenter. The first thing we notice is that the use of remote operations leads the throughput for the different topologies to vary considerably. This happens since remote operations need to go through the causality layer, which means that topologies where causality information has to perform additional hops between datacenters (ring) are more affected than when using tree topologies with lower heights, such as the clique topology (that has the highest throughput). This effect is also visible when using migrations. However, in this case only the migration messages need to go through the causality layer, while subsequent operations are sent from the client directly to the target datacenter, minimizing this effect considerably.

When comparing the use of remote operations with migrations, we notice that, once again, the tree topologies with highest heights are more affected when using remote operations, while the topologies with lower heights actually perform worse when using migrations. This happens because migrations imply clients send extra messages, which cost is expected to be amortized by the subsequent operations over remote datacenters. Due to the simplicity of the simulator, this is not visible in these results. However, when

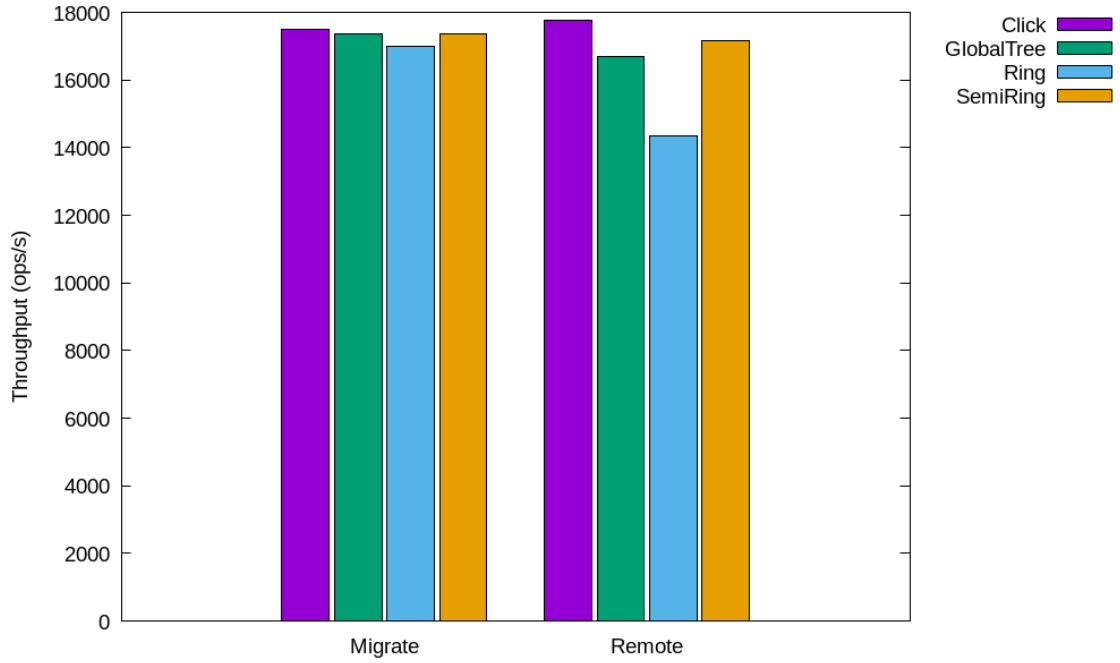


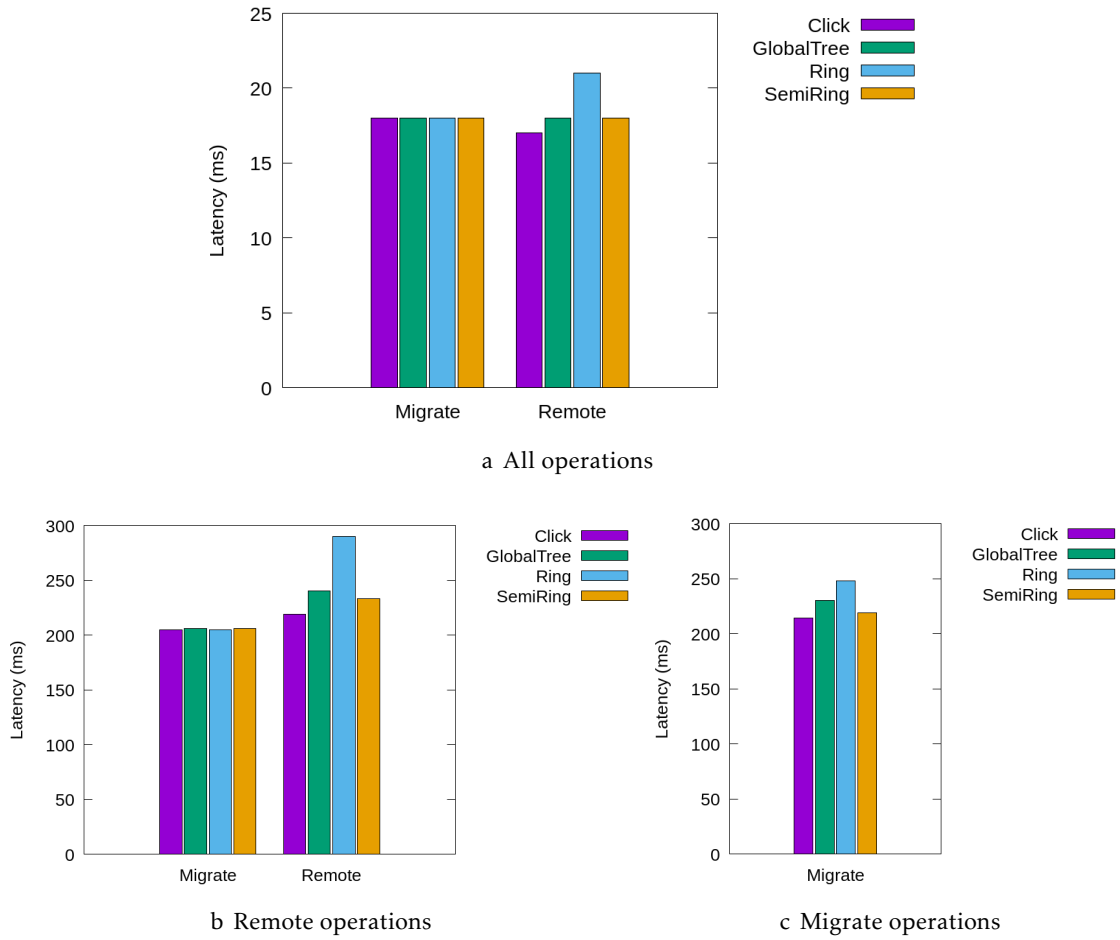
Figure 4.3: System Throughput

we move on to the actual implementation of these algorithms, the difference between remote operations and local operations on remote datacenters will be more significant, since the first will need to use the causality layer while the second will not, which is expected to make this amortization effect noticeable.

4.4.2.4 Client Latency

Figure 4.4, presents the average operations latency as perceived by clients. The response times of local operations are not shown in isolation, since they are constant, being independent from the causality tracking scheme employed. Operations over remote datacenters when using migrations are also constant, since communications happen in a direct fashion between the clients and the datacenter. The most relevant results are the latencies in migrate operations (Figure 4.4c) and remote operations when not using migrations (Figure 4.4b). The first thing we notice is that these results are complementary to the throughput results, which makes sense, since these are the operations that depend on the causality layer and will have the most effect on the achievable throughput of each alternative.

When studying the results reported in Figure 4.4b, we can see that operations over remote datacenters after migrating are faster than remote operations, which is expected. However, the difference is smaller in the average latency observed by clients when considering all operations executed (Figure 4.4a). As explained in Section 4.4.2.3, these extra operations are amortized in subsequent operations, except when using the clique topology. We reinforce that this amortization will happen when using the clique topology in a



a All operations

b Remote operations

c Migrate operations

Figure 4.4: Operation latency as seen by clients

real setting, although our simulation does not capture this effect.

4.5 Lessons Learned

In this section we present the main lessons learned from the previous results, which will were used when moving on to the actual implementation of the algorithm in a real datastore system.

4.5.1 Best Tree Topology

From the results of the simulations we concluded that using the clique topology seems to be the most promising alternative. In retrospective, it makes sense that since we are using a vector clock to keep track of causal dependencies, trees are no longer necessary and, as such, we are only interested in labels reaching their destination as soon as possible.

This solution also has the advantage of being much simpler than using complex trees, which makes the management of the causality layer more simple and efficient. This can

be useful in the future when we start thinking about supporting the dynamic join and departure of datacenters in the system.

We also think that the extra information sent with labels (when comparing to Saturn's propagation) does not impose a significant overhead. On the contrary, it will most likely prove very useful in decreasing data visibility times and maximizing concurrency.

4.5.2 Migrate vs Remote Operations

When deciding whether to migrate clients across datacenters or use remote operations, we decided that migrating is most likely the best choice. As seen on the previous results, this option provides the best results (except in the clique topology where, as we explained, we still expect it to perform better in a real implementation).

Since we also expect the causality layer to be the main bottleneck when executing operations, it makes sense that we try to make as many operations as possible independently from this layer. When we execute a remote read, both the read request and the response need to use the causality layer, which is equivalent to migrating to and back from a remote datacenter. This means that, if a client needs to execute two or more remote operations, migrating should be more advantageous.

4.5.3 Concurrency

While these simulations were relevant in assisting us to understand the implications of some aspects of our causality propagation protocol, there is still an important aspect that we have not covered in simulations, which is the ability to execute operations concurrently.

When using a single global tree, since labels are propagated without any additional causality tracking information, we need to rely only on the label delivery order to know when operations can be executed. As such, we need to assume that every label depends on all previously received labels, which results in operations having to be executed in a serial order.

When using our scheme to propagate causality tracking metadata, the use of the vector clock, besides operation labels, enables us to obtain information to exploit a higher level of concurrency by executing multiple operations in the datastore layer simultaneously. This aspect can be very important from a performance standpoint, especially for datastores that use sharding.

Summary

In this chapter, we presented a study, by simulation, of the alternative topology designs and configurations for our protocol.

In the next chapter, we will present a concrete implementation of our protocol, where we also try to maximize the amount of concurrent operations we can execute, in order to take advantage of sharding in the datastore layer.

Chapter 5

Enriching Cassandra with causal consistency

After having decided which algorithm to implement, and having simulated its execution to understand which variant of the algorithm shows to be more promising, the next step is to implement the algorithm over a real datastore, and to do its experimental evaluation using a realistic setup (with multiple geo-replicated datacenters operating with partial replication).

In this chapter, we start by explaining why we chose Cassandra to implement our solution, discussing some of the relevant implementation details of Cassandra that have impact in our own implementation. We further detail which aspects of Cassandra had to be adjusted in our design. After this, we detail the architecture of our solution and how it interacts with the Cassandra datastore. In the following section we provide the implementation details of our prototype, explaining the changes made to the datastore source code and the implementation of the causality layer. This chapter concludes by reporting the conducted experimental work, followed by the presentation and interpretation of obtained results. We named our prototype *Causal Consistent Cassandra*, or simply C^3.

5.1 Datastore Selection

In order to choose which system to use as the datastore, we need to consider the requirements presented in Chapter 3.1, among others that we detail below. After a brief study of the existing systems, we decided that Apache Cassandra is the best choice for several reasons:

- It supports the requirements presented on Chapter 3.1, more precisely, it implements eventual consistency and can be configured to operate under partial geo-replication

- It supports multiple levels of consistency, since it is possible to execute operations using different consistency configurations in order to have better data freshness (i.e, higher chances of reading the most up-to-date version of each accessed data object). For instance, while executing a read operation, the user can use the consistency level *QUORUM* to read from a quorum of all nodes that replicate the data. Having these several levels of consistency can be useful when trying to guarantee causality since we need to control which nodes are used to fulfill client requests.

- By using a gossip protocol, as explained in Chapter 2.5, every node in Cassandra has information about every other node in the system, including which nodes replicate which data objects. Since the datastore nodes have this information, we don't need to maintain it in our causality layer, thus avoiding extra processing in the causality layer. This is also important to keep the separation between the datastore layer and the causality layer.

- It implements sharding in an automatic and efficient manner. To scale the system, all we need to do is add an additional node to a datacenter and the load of that datacenter will be redistributed to include the new node without the help of any additional configuration changes by the database administrator. The use of a DHT inside each Cassandra datacenter is the basis to support this.

- It is a popular and widely used system meaning that it has been exhaustively tested. Additionally, Cassandra is well documented[1], not only on its setup and configuration, but also on its internals which proved to be invaluable when reasoning about how we should modify it to integrate with our causality layer.

- Being open-source was obviously a requirement, since we need to change its code to integrate the causality layer. However, the code-base being well-maintained and readable, combined with the availability of online talks[2] explaining the code structure were decisive reasons for us to adopt Cassandra.

5.2 Cassandra Internals

In this section we explain how Cassandra operates internally. This is relevant to understand how we can change its operation (and code) to integrate it with the causality layer. We will also discuss the best ways to do this integration while changing the Cassandra logic as little as possible. Since we are only worried about adding causality, we will mostly

[1] https://docs.datastax.com/en/cassandra/3.0/index.html
[2] https://www.datastax.com/dev/blog/deep-into-cassandra-internals

focus on the operation propagation part of Cassandra. This discussion builds on the high level concepts of Cassandra presented in Section 2.5. While presenting the changes to Cassandra, we are guided by the requirements to implement the algorithm discussed the previous chapters.

We start by a high level explanation of how operations are executed, followed by the specific details regarding the execution of read and write operations.

To execute an operation, the client starts by connecting to a datacenter. From that moment on, each operation requested by the client is sent to one of the nodes of that datacenter. The node that receives the operation is called the coordinator node and, as the name suggests, becomes responsible for coordinating the execution of the operation. The coordinator node, using the information about the cluster (gathered using the gossip protocol), propagates the request to every relevant node and waits for a reply from a configurable fraction of them. When enough replies are received from the nodes involved, the coordinator replies to the client, according to the replies received. The nodes to which the coordinator propagates the operation, as well as the nodes for which the coordinator waits to reply, are chosen depending both on the consistency level of the operation (which is set by the client) and the replication strategy for the keyspace where the object manipulated by the operation is located.

5.2.1 Execution of read operations

We now discuss how read operations are executed in Cassandra, followed by the changes required to integrate this execution with our causality layer.

5.2.1.1 In Cassandra

When a coordinator receives a read operation from a client, it starts by deciding from which nodes the data will be read (as stated above, this depends on the consistency level). It then sends a read request to the closest node and a digest request to the others. The digest request is used to decrease network usage, since the node that receives the digest request responds with a digest of the data instead of the data itself. When the coordinator receives the required responses, it returns to the client the response with the most recent data. If the contacted nodes had different versions, the coordinator is also responsible for executing the *read repair* protocol. This protocol is used to make data consistent across all nodes. The coordinator can also contact extra nodes (more than the consistency level requires) to execute the read repair protocol. This is implemented in order to try to maintain the database as consistent as possible as fast as possible.

Figure 5.1 illustrates the execution of a read operation with the consistency level *LOCAL_QUORUM* in a cluster with two datacenters and three replicas in each datacenter. In this situation, node 10 is the coordinator of the read operation and nodes 1, 3, and 6, and 4, 11, and 8 are the ones that replicate the data required for this operation in each of the two datacenters, respectively. Since the consistency level, as the name suggests,

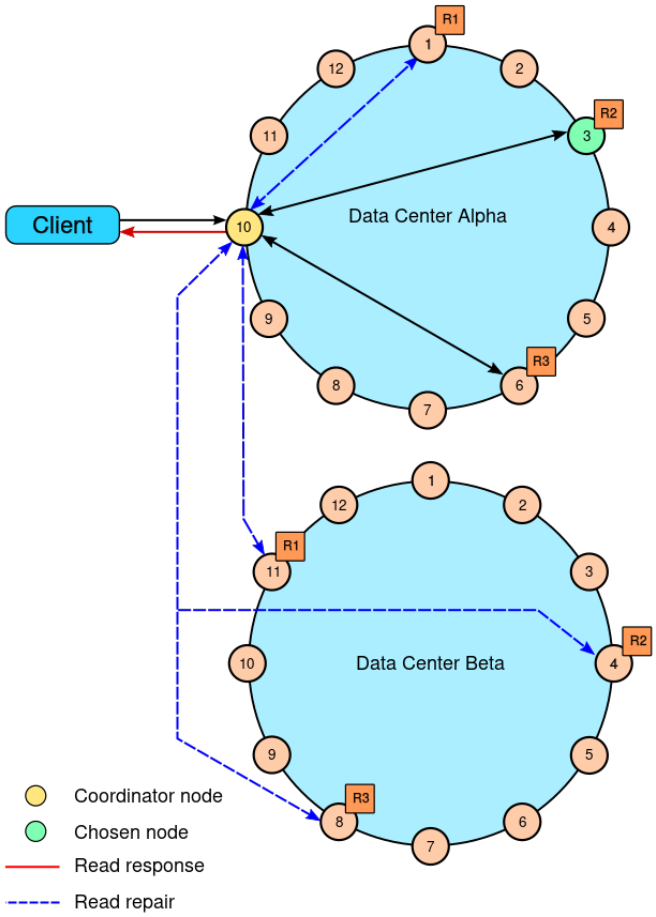


Figure 5.1: A read operation with *LOCAL_QUORUM* consistency - extracted from [16]

only requires a quorum in the local datacenter to respond, the coordinator chooses to use nodes 3 and 6 to complete the request (represented by the black arrow). However, due to the read repair protocol, digest requests were also sent to the other nodes. After nodes 3 and 6 respond, the coordinator can reply to the client with the most recent written value (between those two nodes). After replying to the client, the read repair protocol will continue executing to check if the data in the remaining nodes is consistent with the first two nodes and if not, the inconsistent replicas will be repaired in the background (by updating the outdated value).

5.2.1.2 Adding causal consistency

In order to integrate Cassandra with our layer, we need to be able to control which nodes are chosen by the coordinator to fulfill a read request. We need this because we only want to read from a single datacenter, to guarantee reading a causally consistent value. Also, we need to be careful to avoid the coordinator executing the read repair protocol in nodes located on other datacenters, since it could (easily) result in those nodes receiving data

that has not reached them through the causality layer yet[3]. An example of this can be seen in Figure 5.1, if we consider that a write operation over the same data object was executed by the client before this read operation, but that the write operation has not reached the datacenter *Beta* yet, the read repair protocol could make that write operation visible in *Beta* before being delivered by the causality layer, violating causality. The easiest way to avoid this is to create a new consistency level, which only reads from a single datacenter and avoids trying to repair nodes outside that datacenter. Luckily, the Cassandra code structure allows us to easily add new consistency levels.

To support remote reads we need to use the causality layer to propagate both the read requests and the read responses. We need to do this only in remote read requests (i.e requests from a coordinator in a datacenter to a node in a remote datacenter) and remote read responses, since local reads can be executed immediately without using the causality layer. The simplest way we found to integrate read requests/responses with the causality layer is by intercepting these messages when they are delivered from the Cassandra's component that decodes received messages to the component that executes them. After intercepting the messages, we wait until the label from the causality layer arrives to pass the message to the execution component. Additionally, the coordinator must send a label to the causality layer when sending read requests to nodes in other datacenters, and these nodes must also send a label to the causality layer when responding to the read request.

Combining this interception of messages with the new consistency level, we are able to execute both local and remote read operations in a causally consistent manner. In Section 5.4, we explain in more detail how these changes were made.

5.2.2 Execution of write operations

We now discuss how write operations are executed in Cassandra, followed by the changes required to adapt this execution with our causality layer.

5.2.2.1 In Cassandra

In write operations, and similarly to read operations, the consistency level of the operation defines how many nodes must respond to the coordinator before the coordinator replies to the client. However, unlike the read operations, the coordinator propagates the operation to all nodes that replicate the relevant data object, instead of just to the nodes required for satisfying the consistency level. Another difference between read and write requests is the way the operations are propagated by the coordinator to datacenters other than its local one. For each remote datacenter, the coordinator chooses a single node and sends it the write operation with a special tag for it to forward that operation to its local replicas. During a write operation, even if some nodes are down, the operation is

[3]Notice that some mechanism has to be put in place to deal with lost messages between data centers or operation coordinator failure. Although in our prototype we did not address this, a simple solution would be to rely on a publish-subscribe system with durability, such as Apache Kafka [30], to support communication between the coordinator of the local operation and relevant nodes in remote data centers.

successful as long as enough nodes respond. In order for the temporarily unavailable nodes to catch up when they recover, the system relies on both the read repair protocol explained previously and the hinted handoff repair mechanism. *Hinted handoff* is a repair process used by Cassandra which consists in, when a node is unavailable, the coordinator storing the data to be sent to that node in a set of *hints*. When the node becomes available once again, the coordinator hands off these hints (hence the name) to allow the node to update its state with the missed operations.

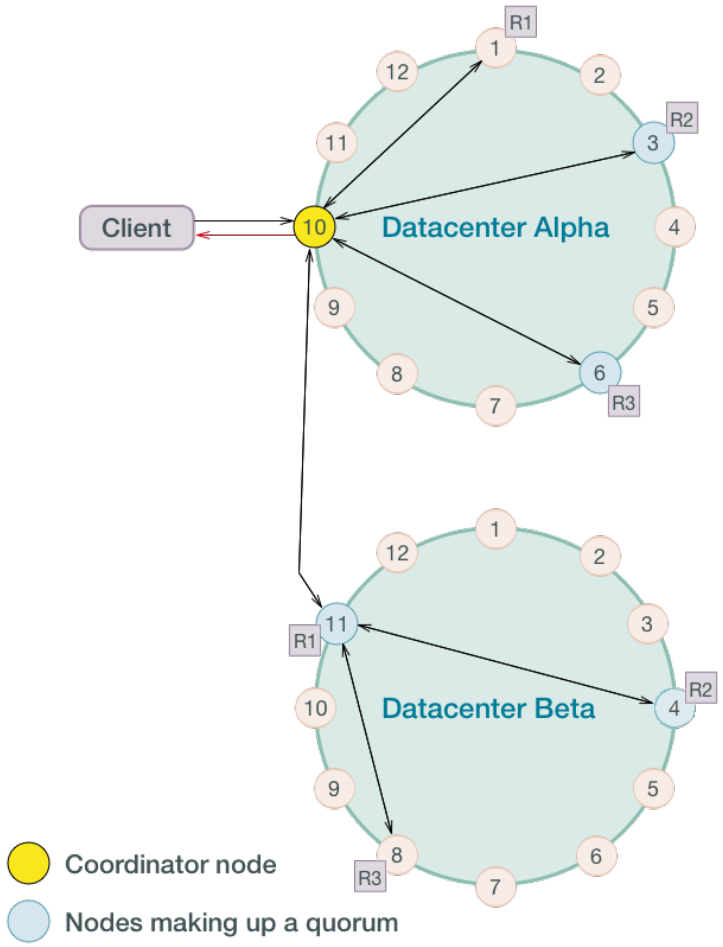


Figure 5.2: A write operation with *QUORUM* consistency - extracted from [29]

Figure 5.2 illustrates the execution of a write operation in a cluster with the same nodes and replication strategy as the previous example. In this situation, the consistency level is set to *QUORUM* which means that a majority of the total number of (individual) nodes must respond to the coordinator. In the figure we can see that the quorum that first responded was composed of nodes 3, 6, 11, and 4 (darker colored nodes). We can also see in the figure, as explained previously, that the coordinator sent the operation to a single node in datacenter *Beta* (node 11), which in turn, forwarded the operation to the other nodes in its local datacenter (nodes 4 and 8). Despite not being represented in the figure, every node responds directly to the coordinator (including nodes 4 and 8).

5.2.2.2 Adding causal consistency

Similar to read operations, in order to integrate write operations with the causality layer we need to make two relevant changes to the operation of Cassandra: create a new consistency level and intercept write requests.

Starting with the new consistency level, we need a consistency level that requires responses from a quorum inside a single datacenter, be it the local datacenter or a remote one (depending on the manipulated data object being replicated in the local datacenter or not). We do not need to modify the number of nodes to which we propagate the operation, since we want to write to all of them anyway. This consistency level will only change which nodes the coordinator needs to wait for in order to be able to produce an answer to the client.

Regarding the interception of operations, in tandem with read operations, we need to intercept write operations before they are executed by Cassandra nodes and wait until the node receives the label from the causality layer before executing it. Unlike read operations however, we also wait for the label while executing local writes. This is required to enforce causal consistency for reads executing concurrently (and without coordination from the causality layer). Write responses, in contrast, don't need to use the causal layer since they are simply acknowledgment messages (unlike read responses which actually contain data), and can be sent directly to the coordinator. Considering that for inter-datacenter writes the coordinator sends only a single message to a node per datacenter, we decided that it makes more sense that each of these picked nodes should forward the operation (without executing it) as soon as possible without having to wait for the causality layer, since otherwise we would be adding unnecessary delays to the data visibility.

By combining these two modifications (the new consistency level and the interception of write requests), we are able to support both local and remote write operations, while always enforcing causal consistency across all operations. In fact local and remote writes behave exactly the same way, with the only difference being that the coordinator waits for the quorum from nodes in the local datacenter or in a remote one. Again, in Section 5.4 we explain in more detail how these modifications were performed.

5.3 Causally Consistent Cassandra Prototype

In this section we present the complete design of our solution after implementing the causality layer and integrating it with Cassandra, by doing the required modifications as discussed in the previous section. In order to simplify this presentation, we divide the system architecture in three components: the client, the datastore layer (modified Cassandra) and the causality layer (that implements our algorithm). We then explain how read, write, and migrate operations are executed in our prototype.

5.3.1 Client

When executing read and write operations in the local datacenter, the client behaves exactly in the same way as if it was executing operations in a regular Cassandra system. This is possible since we don't use any kind of dependency tracking information on the client side, thus making the enforcing of causality transparent for the client. A small exception since when the client needs to execute operations with a new consistency level which did not exist in the original Cassandra, however this does not change the client behavior since the client uses this new consistency level just like it would use any of the existing ones.

Executing remote reads and remote writes is also transparent to the client, since it will be up to the datastore and causality layers to make sure the operations are executed in a causally consistent manner. The client simply waits for the coordinator to reply with the result without being aware of what happens internally in the system.

The transparency is lost for supporting client migrations between datacenters instead of using remote operations. This happens because, in order to migrate to a remote datacenter, the client communicates directly with the causality layer. The client sends a migration request to the causality layer node in its current datacenter, then waits until it receives a response from the target datacenter, at which point it can start executing operations on the new datacenter. It is the causality layer's responsibility to make sure the migration is done while ensuring causal consistency. The migration request created by the client includes information about which datacenter the client wants to migrate to. The information about which datacenters replicate which partitions of the datastore is provided by the Cassandra driver. The client can also attach a list of possible target datacenters, instead of a single one, to the label. This can be useful when multiple datacenters replicate the data that the client is interested in. In this case, the causality layer will choose the datacenter with the lowest load in order to speed up the client migration.

5.3.2 Datastore layer

The datastore layer operates in a similar fashion to the original Cassandra system, but with the addition of the integration modifications presented in Section 5.2.

It propagates client operations in a regular way to the relevant nodes but only executes client operations after receiving the label from the causality layer. The exception is local reads, which can execute immediately without coordination with the causality layer. This layer is also responsible for generating the labels corresponding to each client operation and delivering them to the causality layer. These labels already contain the target datacenters and nodes for the operation, thus avoiding the need for the causality layer to have information about the datastore layer.

By using the consistency level we created, this layer makes sure to only reply to the client after receiving responses from a quorum in a datacenter. For local operations this quorum is always exclusively composed by nodes in the local datacenter. For remote

writes it is the first quorum to respond in any datacenter, and for remote reads it is the quorum in the closest datacenter that replicates the data object relevant for the operation.

5.3.3 Causality layer

The causality layer, as the name suggests, is the layer responsible for tracking and enforcing causal consistency across the entire system. This layer is not aware of the data distribution and replication in the datastore layer, it simply receives labels corresponding to operations and propagates them in a causal manner to the target nodes in local or remote datacenters (note that labels already contain the target nodes of operations).

Each node in the causality layer maintains the following state: an *operation counter*, used to timestamp write operations; an *executed clock*, a vector with one entry per data center recording the timestamp of the latest operation executed from that data center; an *executing clock*, a vector recording the timestamp of the latest operation from each data center in execution in the local data center.

When the causality layer receives a label concerning a new local operation, including a unique identifier and the set of datacenters (and corresponding nodes) where the operation is to be delivered to, it sets the label dependencies to be those of the *executing clock* and, if it is a write operation, increments the local operation counter and uses it to assign a timestamp to the label.

The causality layer puts the label of the new operation in a log of pending operations to execute (in case the object being accessed by the operation is replicated locally), and also propagates it to the causality layers in relevant data centers (i.e., data centers that replicate the accessed object). When a causality layer receives a label from a remote data center, it adds it to the log of pending operations. A pending operation is ready to execute when the operations it depends upon have already completed, i.e., all entries of the *executed clock* are larger or equal to the entries in the dependencies of the operation.

When an operation is ready to execute, the causality layer propagates the label to the local nodes responsible for executing that operation. If it was a write operation, these nodes then acknowledge the causality layer when the execution completes, which allows the causality layer to know when the operation is completed locally and update the *executed clock* accordingly. In conjunction with our mechanism to record the causal dependencies of an operation, this approach guarantees that an operation only executes after all operations it depends upon have completed. A simple clock message is also sent to the remaining datacenters, (i.e., those not directly interested in the operation) that notifies them of operations executed on other datacenters that do not affect their locally stored objects, allowing them to update their *executed clock* accordingly.

Migrations are also completed in the same manner. After receiving a migration label from a client, the *executing clock* is attached to the label and sent to the target datacenter. If the client specified multiple datacenters in the label, the one with the lowest load will be chosen (we explain how we do this later in Section 5.4). When the node in the target

datacenter receives the migration label, it waits until the local *executed clock* matches the label's clock and then responds to the client.

5.3.3.1 Concurrency and the use of two vector clocks

As discussed in Section 3.1, we want to allow a high level of concurrency when executing remote operations. To this end, we need to be able to detect concurrent operations, to enable them to be executed in parallel. To do this, using a single clock in each datacenter is not enough, since that would result in every write operation that was executed in that datacenter to be tagged with a clock position that made it dependent on every previous operation. This would mean that when executing these operations in a remote datacenter, only one could be executed at a time, effectively limiting concurrent execution of remote operations to a single operation per remote datacenter.

The use of the two vector clocks (*executed clock* and *executing clock*), as explained above, was implemented to solve this problem. By doing this, we attempt to maximize the number of concurrent operations that can be executed simultaneously, while enforcing causality. Since we are attaching the *executing clock* to each label, each operation needs to wait for the already executing operations before it starts being executed, this is necessary in order to maintain causality inside each datacenter. In other words, we assumed that each operation is causally dependent on every write operation being executed in the local datacenter. If this was not done, we could risk a client reading the result of an ongoing write operation and then executing its own write operation, which could then be propagated to remote datacenters before the write operation previously observed, which would lead to a causality violation.

5.3.4 Operation Execution

After presenting the design of our solution, we now detail the algorithms for executing operations.

5.3.4.1 Local Read

The local read is the simplest operation in our solution, since we don't need to track any kind of causality information. The algorithm operates as follows:

1. The client sends the read operation to a chosen local coordinator.
2. The coordinator propagates the operation to a quorum of local nodes that have a replica of the object being read.
3. Each node that receives the operation immediately responds to the coordinator.
4. After receiving the response from the quorum, the coordinator replies to the client with the most recent value received from the quorum of nodes.

5. In the background, the *read repair* protocol can be executed, but always only in the local datacenter.

5.3.4.2 Local Write

A local write, while only requiring responses from the local nodes to finish, also needs to propagate the operation to remote datacenters. The execution proceeds as follows:

1. The client sends the write operation to a chosen local coordinator.
2. The coordinator calculates which nodes in the entire cluster replicate the operation's data. It then propagates the operation to all nodes in its datacenter and to one node in each remote datacenter (as explained in Section 5.2.2). It also generates a label and delivers it to the causality layer.
3. The causality layer attaches the *executing clock* and the local operation counter to the label and propagates it to other datacenters.
4. When the local *executed clock* allows it, the label is delivered to the local nodes and the local *executing clock* is updated.
5. After the local nodes receive both the label and the operation, they execute the write and reply to both the coordinator and the causality layer.
6. The coordinator receives a quorum of local responses and replies to the client. At this moment the client finishes executing the operation, but the system is (potentially) still propagating it to remote datacenters.
7. The causality layer receives confirmation that a quorum of nodes executed the write locally and updates its *executed clock*
8. Each remote node in the causality layer that received the label waits until it can execute the operation and then propagates it to its local nodes.
9. Each remote node, after having received the label and the operation (from either the coordinator directly or the chosen node in its datacenter) executes the operation and replies to both the causality layer and the coordinator.
10. After a quorum of nodes in each relevant datacenter finishes executing the operation, it is considered completed.

5.3.4.3 Remote Read

Remote reads are considerably different from local reads, since they need to use the causality layer to coordinate their execution. We note that, while remote read and remote write operations were implemented, we do not report them in the experimental results,

since their use results in lower overall performance when comparing to the use of migrate operations. We still report these operations for completeness. The algorithm execution operates as follows:

1. The client sends the read operation to a chosen local coordinator.
2. The coordinator chooses the closest datacenter that replicates the data and propagates the read operation to a quorum of nodes in that datacenter. It also generates a label and delivers it to the causality layer.
3. The causality layer attaches its *executing clock* to the label and propagates it to the remote datacenter.
4. The causality layer node in the remote datacenter receives the label and waits until it can execute the operation (i.e, until its *executed clock* matches the label's dependencies). It then propagates it to its local nodes.
5. Each node in the remote datacenter, after having received both the label and operation, responds to the coordinator and also delivers a label to the causality layer. These labels are tagged with the remote data center's *executing clock*.
6. The response labels are propagated to the client's local datacenter, and are delivered to the coordinator when the causality layer node's *executed clock* allows it.
7. When the coordinator receives both the responses and the responses' labels, it can reply to the client.

5.3.4.4 Remote Write

The sequence of steps in the execution of a remote write is very similar to a local write, since all nodes need to receive the operation anyway, the key differences are that the local datacenter does not execute the operation and, consequently, the coordinator must wait for a remote quorum of responses.

1. The client sends the write operation to a chosen local coordinator.
2. The coordinator calculates which nodes in the entire cluster replicate the operation's data. It then propagates the operation to one node in each remote datacenter (as explained in Section 5.2.2). It also generates a label and delivers it to the causality layer.
3. The causality layer attaches the *executing clock* and the local operation counter to the label and propagates the label to remote datacenters.
4. Each node in the causality layer that receives the label waits until it can execute the operation and then propagates it to its local nodes.

5. Each remote datastore node, after having received the label and the operation (from either the coordinator directly or the chosen node in its datacenter) executes the operation and replies to both the causality layer and the coordinator.

6. When the coordinator receives the first quorum of responses from one of the datacenters, it replies to the client. At this moment the client has finished executing the operation.

7. After a quorum of nodes in each relevant datacenter finishes executing the operation, the operation is considered completed.

5.3.4.5 Migrate

The migrate operation is executed between the client and the causality layer, without having to go through the datastore layer. The execution is as follows:

1. The client wants to execute an operation over a keyspace that is not replicated locally. It sends a label to the causality layer with either the datacenter it wishes to migrate to or a list off all datacenters that replicate the intended data object (i.e, the target keyspace).

2. The causality layer receives the label. If the label contains a list of datacenters, it chooses the one with the lowest load as the target. It then attaches its *executing clock* to the label and propagates the label to the target datacenter.

3. The causality layer node in the target datacenter receives the label and, when its *executed clock* allows it, answers to the client.

4. The client receives the label from the remote datacenter and, from that moment on, it can start executing operations directly on it.

Another option could be for the client to send the migrate operation to the datastore layer, just like when executing any other operation. However, for simplicity's sake we decided to send it directly to the causality layer, since the other option would mean modifying both the Cassandra driver's source-code executing in the client and the datastore to support the migration operation.

5.4 Implementation details

In this section, we present some implementation details on our prototype. We start by explaining how the Cassandra source code was changed to implement the modifications required for the integration with the causality layer. We divide this in two parts: the creation of the new consistency model and the inter layer communication. After explaining the changes to Cassandra, we then present the details of how we implemented the causality layer and how it operates.

5.4.1 Consistency Model

Since Cassandra already supports multiple consistency levels, the source code related to consistency is structured in a generic manner, which means that creating our custom consistency model was reasonably simple. We started by examining the consistency level *EACH_QUORUM* since it is the one closest to what we want. *EACH_QUORUM*, as the name suggests, requires operations to be executed in a quorum in each datacenter before replying to the client. Since for our consistency level we need to execute the operation in a quorum of a single datacenter, we concluded that we could adapt *EACH_QUORUM*'s code to create it. We decided to name our consistency level *ANY_QUORUM*.

We now present each rule that a consistency level needs to implement in Cassandra, and explain why it is required and how we implemented it.

BlockFor: This rule is used to know the minimum number of responses an operation needs to wait for before returning to the client. This is useful in many situations, for instance to check if we have enough nodes available to execute the operation, or to know how many nodes failed to respond when an operation times out. Since we need responses from a quorum in a single datacenter, we implemented this rule by calculating the size of a quorum of nodes replicating the data in the closest datacenter (in local operations, the closest datacenter is the local one). This rule is valid for all operations: local and remote reads and local and remote writes.

FilterForQuery: This rule is used by Cassandra to decide which nodes in the cluster are going to receive a read operation (since write operations always go to every node). We implemented this by first finding out the closest (in terms of latency) node that replicates the data we're interested in. We then check which datacenter that node belongs to (if the data exists in the local datacenter, it will always be the chosen one) and select a quorum of nodes in that datacenter to receive the operation.

AssureSuficientLiveNodes: This rule is used to check if there are enough live nodes in the cluster to execute the operation with the required consistency level. This is useful to prevent starting the execution of operation that we already know cannot be completed. For some consistency levels this is trivial as we only need to compare the number of total nodes alive with the rule *BlockFor*. In our case we implemented this rule by checking if any datacenter has enough nodes alive to fulfill the operation.

ResponseHandler: This rule is used to check whether enough responses have already been received by the coordinator. This is only used for write operations, since in read operations we simply wait for the number of responses chosen by the *BlockFor* rule. The way we implemented this rule is by storing how many responses we received from each datacenter and then, each time a response is received, verifying if we have a quorum of responses from any datacenter.

It's important to note that all these rules are generic enough to support all operations (remote and local reads as well as remote and local writes).

Apart from defining the rules for our new consistency level, we also had to make two additional small changes. We disabled the inter-datacenter read repair protocol when using our consistency level since, as explained in Section 5.2, we do not want to risk it breaking causal consistency.

We also needed to make a very small change to the Cassandra driver in order to allow using our consistency level, which consisted in added our consistency level to the Cassandra driver. This was needed since, when configuring the client's consistency level, the driver checks if that level exists by verifying a static list of allowed consistency values.

5.4.2 Inter layer communication

In this section we explain how the Cassandra code was modified to allow it to communicate with the causality layer. This consists in three steps: the coordinator generating labels to send to the causality layer; a node storing a received operation and waiting for the label to arrive from the causality layer (or vice-versa); and a node sending an acknowledgment message to the causality layer confirming that a write operation was executed.

5.4.2.1 Label generation

In order to support propagating operations to remote datacenters, we need to find where, in the code, inter-node messages are generated. We cannot just generate labels for every single message since we only want to add causality support for read and write operations, and do not want to change the behavior of other Cassandra specific protocols (like the Gossip or the schema propagation protocols).

We consider four specific situations where we want to generate labels in addition to simply sending the normal message: when the coordinator propagates a write, regular read, or range read operation and when a node responds to a read operation. A regular read differs from a range read since the former only needs data from a single key, while the latter reads from a range of keys. After finding the code locations dealing with these situations, we modified them in the following way:

1. The node sends each message to the target nodes (or single node in the read response case) as it would normally. However, if the messages require coordination with the causality layer, it attaches a special tag to them symbolizing this. Additionally it stores the message identifiers and the target nodes in a list.

2. After sending all the messages, the node creates a new label, to which it adds the type of the message(s) sent (read, write, or read response) and the list were it stored the message identifiers and the nodes that received them.

3. That label is sent to the causality tracking layer, which will be responsible for splitting it and delivering it to the correct nodes.

This logic was implemented in the same way in all four cases, with a single exception: since all write operations (even local writes) use the causality layer, write messages are tagged and their information is stored on the list even when not sent to remote datacenters.

5.4.2.2 Message interception

After having implemented the label generation, we now need to implement the label receiving logic. We start by identifying the region in the code where the network module of Cassandra decodes received messages and hands them to the module responsible for executing operations. After locating this code, we created an additional module which will be responsible for holding received messages until their labels arrive (or vice-versa) and modified the network module to deliver messages to our module instead of the execution module.

With this done, we are now intercepting every message received in each node, both labels from the causality layer and regular Cassandra messages from other nodes. The next step is to filter which messages we want to hold. Since we're only interested in specific messages (the ones with the tag symbolizing the need to wait for the label), every other message (such as Gossip or schema propagation messages) will be immediately delivered to the execution module.

For the messages that require labels, the behavior is simple. We keep a table for all the received labels and another for the received messages and use them in the following manner:

When a label is received from the causality layer and the corresponding message is already present in the messages table, we can remove it from the table and deliver it to the message execution module. If the corresponding message is not present we store the label in the labels table.

When we receive a message from another node we execute the complementary protocol. If the corresponding label is present, we remove it from the table and execute the message. If the label is not present we store the message in the messages table.

5.4.2.3 Acknowledgment

Since the causality layer needs to know when an operation has been successfully executed in each datacenter, each node in the datastore layer is also responsible for generating a special *Acknowledgment* message and sending it to the causality layer each time it completes a write operation. This information will then be used by the causality layer to detect when a quorum has completed an operation and increase its *executed clock*, enabling it to deliver more labels if there are any pending.

5.4.3 Causality layer

The causality layer is composed by a single process (causality layer node) running in each datacenter, with all nodes being able to communicate with each other.

Each of these nodes is divided in two components: a *receiver* and an *executor*.

Receiver: This component of the causality layer is responsible for receiving and handling messages from the local datastore layer. When it receives a label from the datastore (which can correspond to a write operation, a migration, or a remote read operation), it attaches a copy of the current *executing clock* to that label (and the operation counter, if it is a write operation) and delivers it to the executor components in every relevant datacenter (including its own). When an acknowledgment message is received, it checks if the corresponding write operation has been acknowledged by a quorum of (local) datastore nodes and, in case it has, marks the operation as completed and updates the local *executed clock*.

Executor: This component is responsible for storing all waiting labels and checking when the corresponding operations can be executed. When the local *executed clock* allows a label to be executed, it propagates that label to the relevant target(s), which can be a client, if it was a migration label or local datastore layer nodes, if it was a write or remote read operation directed at the local datacenter. In order to more efficiently organize waiting labels, this component maintains a queue with waiting labels for each datacenter (including the local datacenter). Since labels are propagated among instances of the causality layer in FIFO order, this component only needs to look at the head of each of these queues when checking if it can execute pending labels, since every other label in a queue either depends or is concurrent with the label currently at the head position.

5.4.3.1 Status Messages

As a small performance enhancement, we implemented a simple status propagation protocol. Periodically, each causality layer node sends a message to remote causality nodes informing them of the status of its local queue of pending labels. This information is then used to direct client migrations to the datacenter with the lower amount of load (when the client specifies multiple possible target datacenters in its migration message). This allows to speed up migrations which has an impact on the overall performance of the system.

5.4.4 Saturn

In addition to implementing our solution prototype, and in order to be able to compare our solution with Saturn, we also implemented a version of Saturn on top of Cassandra. Since Saturn requires changing the behavior of both the datastore layer and the client,

instead of just intercepting messages and generating labels, we needed to add a few more changes to Cassandra, apart from ones similar to the changes presented in this section.

Following the design presented by [7], we leverage on the Cassandra's coordinator to materialize the *frontend*, the Saturn's component that hides from the client the datastore internals. This was done because the coordinator already mediates the access of the client to the data center internals. To materialize Saturn's *gears*, the component responsible for generating and manipulating operation's labels, we use Cassandra's nodes by changing their behavior to generate a label for each received write operation. These labels are generated taking into account the client's label (which is attached to the write operation) and are then propagated to the causality layer. Moreover, the write operation mutation was modified to write the label to the datastore with the modified data, which is essential to ensure that the label can be sent to clients that read the object in the future.

To implement the components *label sink* and *remote proxy* of Saturn, the causality layer components that are responsible for receiving and delevering labels from and to the datastore, we modified our own causality layer to implement the Saturn logic. All our changes were performed to try to strictly respect the logic and algorithms described in [7]. Still, we note that we have not enforced linearizability inside each data center, relying instead in a weaker consistency model. The implication of this is that our implementation might be more efficient than a correct Saturn implementation at the cost of not correctly enforcing causal consistency in all situations.

5.5 Experimental Work

In this section we present an evaluation of our work. We start by discussing the employed experimental setup and configurations, clarifying the decisions we made.

5.5.1 Setup

To run our experiments, we used the cloud platform Microsoft Azure[4]. Since we wanted to test a geo-replicated setting, we created multiple datacenters (9 in total) spread across the world in the following locations: Southeast Asia, South Brazil, Central Canada, West Europe, Central India, East Japan, East US, West US and Southeast Australia. Table 5.1 shows the measured latencies between these datacenters. Since we also wanted to measure the effects of sharding, we needed multiple Cassandra nodes in each datacenter. As such we used four virtual machines in each of these datacenters, each running a Cassandra instance. The used machines are of the type *A2 v2*, each having two CPU cores, 4 gigabytes of ram and an hard disk drive with 20 gigabytes. The virtual machines CPUs vary between one of three models in each datacenter (something we could not control) and are the following: *Intel(R) Xeon(R) CPU E5-2673 v4 @ 2.30GHz*, *Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz* or *Intel(R) Xeon(R) CPU E5-2673 v3 @ 2.40GHz*. All of them

[4]https://azure.microsoft.com

however, are sufficiently similar among them and hence, should have negligible impact on our results.

	EUS	JAP	AS	AUS	IND	CA	WUS	EU	BR
EastUS	-	154	221	206	190	27	88	133	122
Japan	155	-	72	121	129	169	106	235	263
Asia	221	72	-	95	62	234	172	187	331
Australia	206	121	96	-	152	226	185	287	314
India	190	128	63	153	-	202	228	128	298
Canada	27	169	234	225	203	-	67	96	133
WestUS	87	106	172	181	228	65	-	157	188
Europe	133	235	186	287	127	96	157	-	192
Brazil	122	262	330	314	298	134	188	192	-

Table 5.1: Latencies between experimental evaluation data centers (ms)

Regarding the partial replication aspect, we created 9 data partitions, one named after each datacenter (which acts as the main datacenter for that partition) and replicated them across multiple datacenters. The number of datacenters which replicates each partition varies between 3 and 5. Each partition is replicated in the datacenter it was named after, at least one datacenter near it and then in one or more random datacenters. Inside each datacenter, each partition is also replicated in 3 of the 4 existing nodes. Table 5.2 shows the distributions of partitions across the datacenters.

Partition	EUS	JAP	AS	AUS	IND	CA	WUS	EU	BR
EastUS	x					x			x
Japan		x	x	x	x			x	
Asia		x	x	x	x				x
Australia	x	x		x					
India			x		x			x	
Canada	x					x	x	x	x
WestUS	x		x		x	x	x		
Europe			x			x	x	x	
Brazil	x							x	x

Table 5.2: Partition distribution across data centers for the experimental evaluation

5.5.2 YCSB

In our experiments, we used *Yahoo! Cloud System Benchmark*[10] to emulate multiple clients executing operations in the system and to gather performance metrics for these experiments.

For simulatin the migration of a client, we adapted YCSB to support the migrate operation (and halt the execution of operations while a client is migrating). We also added some extra logic to handle when a client should migrate and how many operation it should

execute in each datacenter. These changes are only used when running experiments using either our protocol or Saturn (and not when running the baseline Cassandra system).

Since Saturn assumes the existence of a client library that manages a label associated with the last operation of each client, we also had to adapt YCSB to emulate this client library behavior by handling the logic of maintaining, updating and attaching the client label to operations.

5.5.3 Experimental Parameters

To study how our solution compares to others, we ran experiments using five different configurations, where three of them use the baseline, eventual consistent Cassandra datastore, with different configurations and hence, providing different consistency properties:

Cassandra with local quorum consistency (*E-LQ*): This configuration uses a regular Cassandra cluster, with eventual consistency where clients execute operations locally using the consistency level *LOCAL_QUORUM*, which means they only need to wait for the response of a quorum of local nodes (i.e two out of three). When the client needs to execute operations over remote data, it uses the consistency level *TWO* (which needs to wait for any two nodes to reply), which should in the large majority of cases result in a quorum in the closest remote datacenter. This is the closest way to achieve the wanted behavior of clients reading from a single datacenter.

Cassandra with quorum (*E-Q*): When using this Cassandra configuration, which also offers eventual consistency, clients use the consistency model *QUORUM* for all their operations, which means they need to wait for a response from a quorum (majority) of the total number of nodes that replicate the data object accessed by the operation.

Cassandra with each-quorum (*E-EQ*): In this configuration, again using regular Cassandra with eventual consistency, clients use the consistency level *EACH_QUORUM* for write operations, which means they need to wait for a quorum in *each* datacenter where the target data object is replicated before responding to the client. When issuing read operations, they use the consistency level *LOCAL_QUORUM* (or *TWO* for remote reads, which is equivalent).

Causally consistent Cassandra (*C-C*^{3}): This configuration uses the solution proposed in this work and our prototype described in this chapter.

Saturn (*C-SAT*): This configuration uses our implementation of Saturn.

The multiple configurations of regular Cassandra are used to gather results about its performance using different consistency levels that attempt to get close to the causal guarantees provided by our solution.

When using either of the solutions providing causality, one of the machines in each datacenter needs to be running the causality layer process. Since we don't want that

machine to be unbalanced in terms of load, we decrease the number of Cassandra tokens (abstraction used to control the portion of data replicated in each node) in that machine, which reduces the amount of data it replicates, thus decreasing its load.

For the Saturn configuration, we tried to run the centralized algorithm developed by the authors that would generate the optimal tree for our setup. However, due to the high number of datacenters in our setup, the algorithm would take too much time to complete (multiple weeks). As such, we generated a tree ourselves which attempted to minimize the overall latency between all (neighboring) datacenters. While this tree is probably not optimal, we believe that this will have little effect on the results since, as we will explain later, the visibility times in Saturn are affected by the lack of concurrency of operations, and not by the latency between datacenters.

As for the client configuration, we created four extra virtual machines in each datacenter (36 total), with the same specifications as the ones that run the datastore. We then run an instance of YCSB in each of these machines with a variable number of client threads which ranges from 50 to 350, in steps of 50 (in total this varies the total number of clients between 1,800 and 12,600). The number of operations executed by each YCSB instance is always 25,000 (for a total of 900,000 operations), which is then divided by the number of client threads running. Each client executes an equal number of read and write operations, following a zipfian distribution for selecting the object that is targeted by each individual operation.

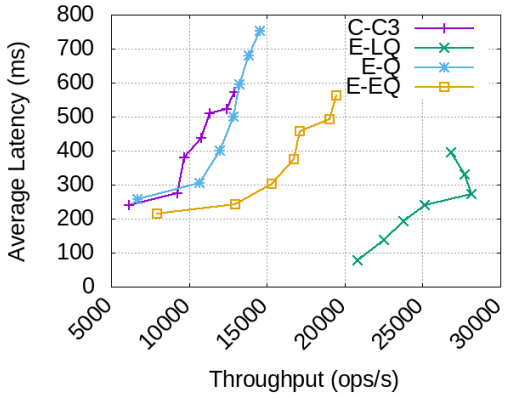
We also tested two different client patterns: issuing only local operations or issuing both local and remote operations. The first pattern consists on clients only issuing operations over data that is available in the local datacenter, which means there is no need for migrations or remote operations. When using the second pattern, clients execute both local and remote operations. In this case, clients sequentially alternate between executing operations on their local datacenter and migrating to some other datacenter and executing operations there. The number of local and remote operations executed are decided by generating Poisson-distributed random numbers, using the lambda values of 95 and 5, respectively, resulting in 95% of local operations and 5% of remote operations. While we implemented both migrations and remote operations in our solution, we concluded, as explained in Section 4.4.2, that using migrations has more advantages and as such, we will only use migrations on our experiments.

5.6 Results

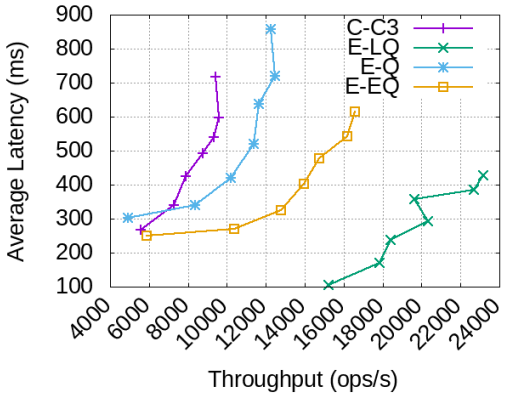
After explaining all the setup and configuration involved in setting up our experimental work, we now present and discuss the results obtained. The main metrics studied are the following: operation throughput, client latency, and data visibility. In the following results, the experiments done with the baseline Cassandra are used mostly as reference points to study the overhead introduced by enforcing causal consistency. These results are not directly comparable with both our solution and Saturn, since these offer different

(and strictly stronger) consistency guarantees. The most relevant comparison is between our solution and Saturn.

5.6.1 Performance versus multiple Cassandra configurations



a Throughput and Latency with only local operations



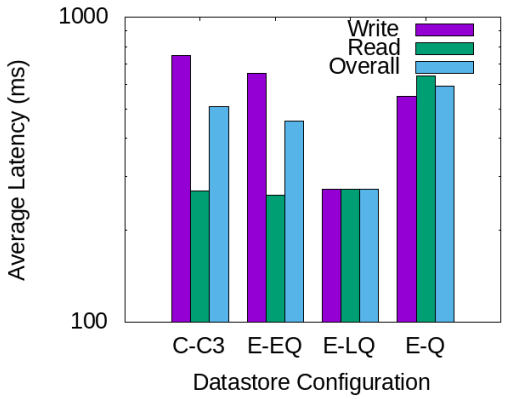
b Throughput and Latency using local and remote operations

Figure 5.3: Performance comparison between our solution and multiple Cassandra configurations

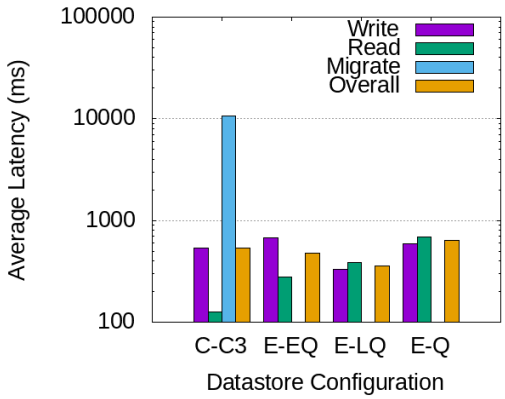
Figure 5.3 reports the overall throughput of the system and client perceived latency when comparing our solution with different Cassandra configurations, for different numbers of clients. Each point in each line represents the (increasing) number of clients executing operations simultaneously that ranges from 1800 (first point in each line) to 12600 (last point in each line). Better results are represented by lower and rightmost points, since these represent lower latency and higher throughput values, respectively. Figure 5.3a represents clients executing only local operations while Figure 5.3b represents the results obtained with clients executing both local and remote operations.

The results are not surprising, with the best results visible when the client only needs a response from the local data center (E-LQ), and worsening as more and further nodes need to be contacted (E-Q and then E-EQ). As expected, due to the overhead of providing causal consistency guarantees, our system shows lower performance than the different Cassandra configurations, albeit we consider this an adequate cost for the additional consistency guarantees.

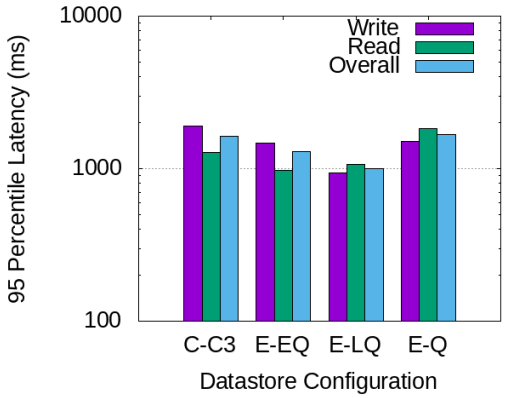
Figures 5.4a and 5.4c report the average and 95 percentile latency of operations as perceived by clients while using only local operations while Figures 5.4b and 5.4d show the same results regarding experiments with both local and remote operations. These figures report the results of the experiments with 9000 clients (corresponding to the fifth point in Figures 5.3a and 5.3b). We selected this data point because there is significant load but the system is not saturated, achieving maximum throughput (for both our solution and Saturn, as confirmed by results reported further ahead in Figures 5.5a and 5.5b).



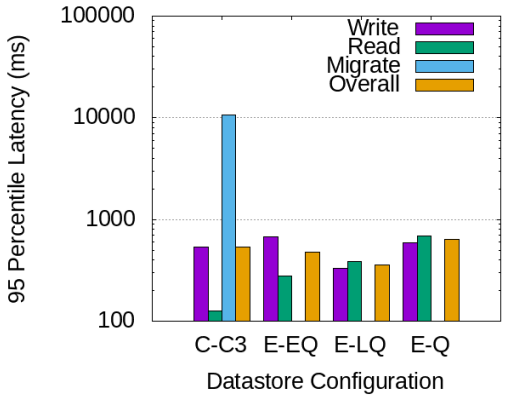
a Average latencies with only local operations



b Average latencies with local and remote operations



c 95 percentile latencies with only local operations



d 95 percentile latencies with local and remote operations

Figure 5.4: Latency comparison between our solution and multiple Cassandra configurations

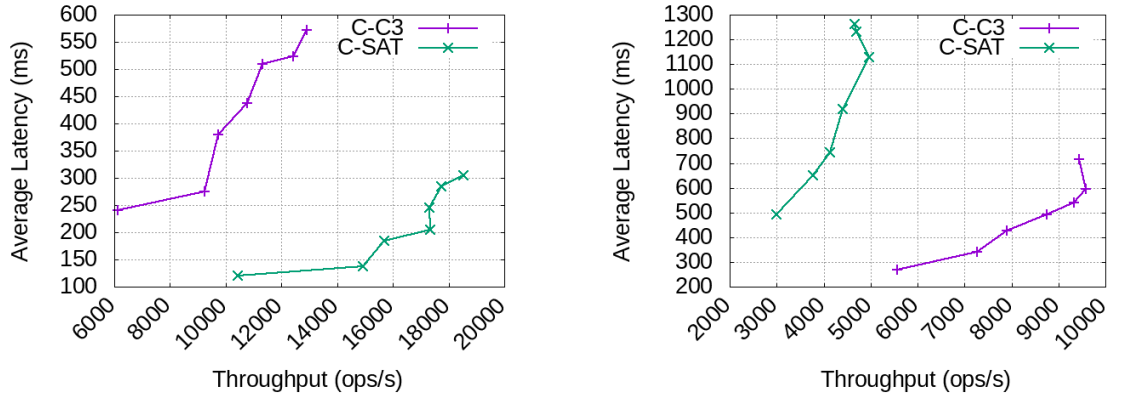
Annex I contains the figures regarding the other data points. Note that the Y axis has a logarithmic scale to make the plots more readable.

Local read latencies are very similar across all cases, except for the configuration of Cassandra using (global) quorums (Figure 5.4a). This makes sense since in all other four experiments the client reads from a local quorum without any coordination in the causal systems (including in our solution), while in this particular one it must read from a global quorum. In write operations, the Cassandra configurations have latencies proportional to the number and distance of nodes needed for gathering the quorum, leading to the expected result that the local quorum has lower latency, followed by quorum and then each quorum. Our solution has a higher write latency, not because of the nodes that effectively need to be contacted but because every write operation needs to coordinate in the causality layer.

Regarding the latencies with both local and remote operations (Figure 5.4b), Cassandra results are similar to the previous ones since the execution of remote operations

follows the same logic as local operations, the only difference being the latency between the client and the remote data center. Our solution's behavior, however has changed considerably since now the client needs to use migrations before issuing operations in another data center. Overall, this means that read latencies in our solution will be smaller, since the datastore nodes have less load, but migrate operations take a longer time to complete, as they need to wait for all causally related write operations to be applied on the remote data center.

5.6.2 Performance versus Saturn



a Throughput and Latency with only local operations

b Throughput and Latency using local and remote operations

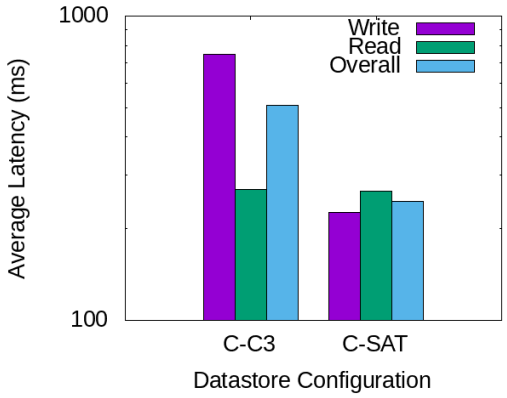
Figure 5.5: Performance comparison between our solution and Saturn

Figures 5.5 and 5.6 represent the same measurements as Figures 5.3 and 5.4 with the difference being that we are now comparing our solution with Saturn.

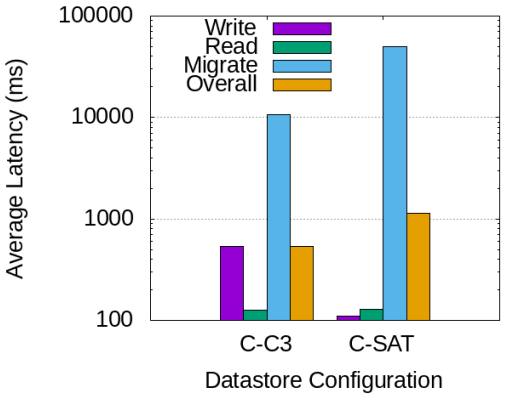
Starting with the throughput versus latency plots, in the local only scenario (Figure 5.5a) Saturn apparently has a much better performance than our solution. This happens because Saturn can execute local write operations faster since it does not coordinate with the causality layer before executing those operations locally. However, as we will show later in this section (Figure 5.7), this also causes it to be very inefficient in executing remote operations. The fact that operations in Saturn don't need to go through the causality layer also means that, in some particular cases, causality may be violated. If a client executes a local write and then another client reads the result of that operation and executes another local write operation, it is possible that the second write (which is causally dependent of the first write) will be propagated through the causality layer before the first one, which will result in a clear causality violation (we recognize that this situations can only happen in extreme asynchronous periods within a data center or in failure scenarios).

In the scenario with remote operations (Figure 5.5b), our solution shows much better performance than Saturn, with Saturn's performance dropping drastically. This happens

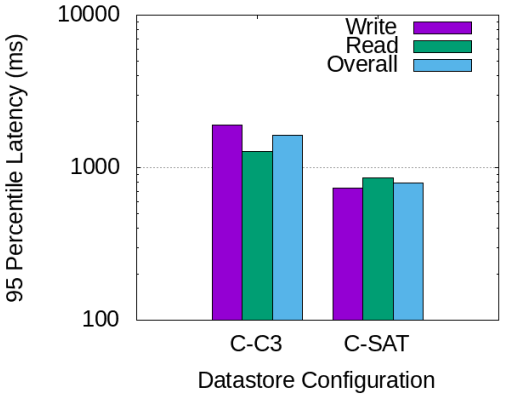
because clients now need to use the causality layer in order to migrate between data centers. Since migration messages need to be propagated through the causality layer and wait for previous remote write operations to complete, the slow rate at which Saturn executes the remote write operations means that clients' migrate operations are en-queued behind a large amount of remote operations, thus needing a significant amount of time before they can be completed, leaving clients inactive for long periods of time. In contrast, in our solution, since we have more metadata and can execute much more remote operations concurrently, we avoid long remote operations queues hence, migration operations are much faster, avoiding clients that need to migrate from remaining stalled for long periods of time.



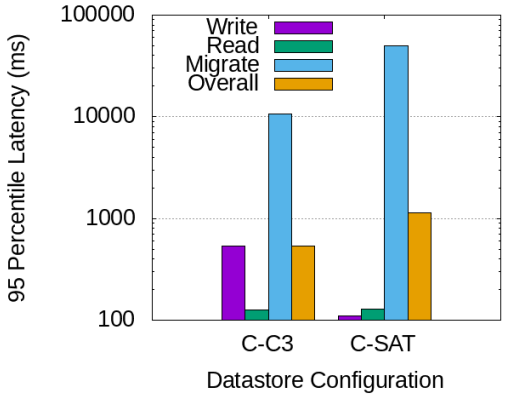
a Average latencies with only local operations



b Average latencies with local and remote operations



c 95 percentile latencies with only local operations



d 95 percentile latencies with local and remote operations

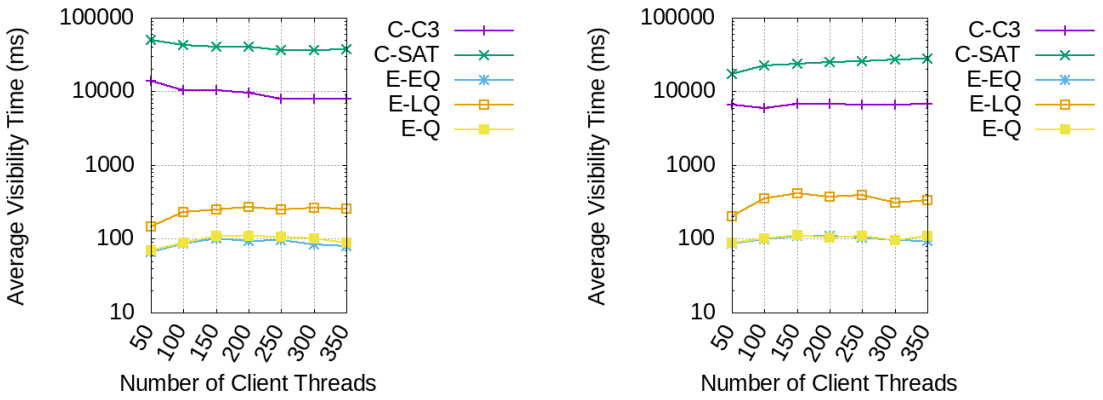
Figure 5.6: Latency comparison between our solution and Saturn

Looking at operation's latency, and starting with the local only scenario, Figure 5.6a shows a big gap in write operation latencies between the two systems. As explained previously, this is due to local write operations in Saturn not coordinating with the causality layer before being executed. Again, while this may make Saturn look better than our solution, it penalizes Saturn when using both local and remote operations (and may not

guarantee causality in some particular cases).

Looking at the local and remote operations scenario (Figure 5.6a), we can see, just like in the throughput versus latency figures, a very different scenario. Writes have lower latency in Saturn since, as previously mentioned, they do not coordinate with the causality layer. The most interesting thing to note, however, is that the latency of migrate operations is particularly high in Saturn. This happens because it is the only operation that needs to go through the causality layer before the client receives a response, instead of just being executed in the local data center. The difference between our solution and Saturn is very significant (remember that the Y scale is logarithmic), leading to the overall latency to be higher in Saturn. As explained before, this happens due to the very slow execution of remote writes in Saturn, which results in very long queues of remote operations and migrations waiting to be executed in remote data centers. This explains the performance of Saturn in Figure 5.5b.

5.6.3 Visibility Times



a With only local operations

b With both local and remote operations

Figure 5.7: Visibility times of each datastore configuration

During the previous discussion, we mentioned the difference in the execution time of remote operations between our solution and Saturn. In Figure 5.7, we report these values. Results show the average visibility time (i.e how long it takes for a write operation to be executed in every data center) for each datastore configuration. Again, note that the Y axis has a logarithmic scale. Table 5.3 reports the raw values for one of these experiences.

Since in both our solution and in Saturn, visibility times are not affected by clients executing remote or local operations, results are similar in both figures. We can, however, see a considerable difference between Saturn and our solution, which justifies our previous arguments that Saturn's visibility times are negatively affected due to limited concurrency in the execution of remote writes which, in turn, negatively affects the overall performance of the system.

#Clients	C-C3	C-SAT	E-EQ	E-LQ	E-Q
50	6,656	17,244	86	200	87
100	5,881	22,446	99	353	100
150	6,765	23,870	108	415	112
200	6,772	24,905	108	372	105
250	6,663	25,481	104	390	110
300	6,688	27,050	95	306	97
350	6,867	27,991	94	333	110

Table 5.3: Raw average visibility time (ms)

5.7 Results Analysis

Having presented and studied the results of our experimental evaluations, we now present the main lessons the can be taken from them.

When comparing with Cassandra, our solution naturally has worse performance, since there has to be a cost for providing stronger consistency guarantees (causal consistency instead of eventual consistency). However, we think that this cost is reasonable, especially when comparing with the Cassandra experiments that use the *EACH_QUORUM* consistency level, which can be seen as the baseline that is closest to the causal consistency experiments (provided by C^3 and Saturn). We also believe that there is still wide room for improvement in order to reduce this overhead, by optimizing our implementation of C^3.

When comparing to Saturn, since in most cases the migration of clients might be inevitable when using partial replication, our approach of balancing the execution of local operations with remote operations shows much better overall results, staying behind only in the latency of the execution of local operations. We also consider our solution to provide stronger causal consistency guarantees, since, as we explained in Section 5.3.3.1, we are not sure that Saturn is able to keep causality in some particular scenarios. This leads us to conclude that having additional causality-tracking information between operations is a very important factor for protocols providing causal consistency, as it enables more parallel and faster execution of remote operations.

It is also worth noting that, due to the inefficiency of Saturn in executing remote operations concurrently in a datastore that supports sharding, if we scale the datastore layer even further, by increasing the number of nodes and increasing the number of clients and operations being executed, it is predictable that the difference in results between our solution and Saturn will become even more noticeable.

Summary

In this chapter, we presented the implementation of our solution over the Cassandra datastore, explaining all the relevant inner working of Cassandra and the modifications

required to adapt it to work with our solution. We then presented the implementation details of our solution, explaining the techniques used to track and enforce causality. We finished by showing the experimental work conducted, along with the analysis of its results and the relevant lessons taken from it.

In the next chapter we conclude this thesis, by first showing our conclusions and then presenting some possible future work.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The causal consistency model is a very appealing consistency model since it occupies a *sweet spot* between weak consistency and strong consistency, by combining the availability and low latency advantages of traditional weak consistency models and providing some consistency guarantees, which are enough to simplify the work of programmers developing applications on top of these datastores.

In this thesis, we presented an in-depth study of the challenges faced when attempting to create a protocol capable of supporting causal consistency in a partial, geo-replicated setting. We exploited a novel approach to achieve causality, introduced by the Saturn system, which consists in separating the datastore layer from the tracking of causality, which allows the causality layer to work with smaller pieces of data and hence lower overhead. We then considered and studied, with the help of a simulator created as part of this thesis, multiple possible approaches to the tracking of causality using this separation of layers. As a result of this study, a novel tracking protocol was designed, created while taking in mind a partially, geo-replicated and scalable underlying datastore, and requiring as few as possible changes to the datastore system.

In addition to the design of this protocol, this thesis also presents a concrete implementation of it. We implemented our protocol over a popular, eventual consistent datastore, Cassandra, describing the changes required to implement our solution on top of this system. Experimental work was then conducted, using a realistic, partial and geo-replicated test setting, in order to validate this protocol, and compare it with both the unmodified datastore and another causality tracking solution, Saturn.

The experimental results show that our protocol is capable of maintaining a good balance between the execution of local and remote operations, by using just enough

metadata to allow concurrency in the execution of remote operations, which can be very helpful when scaling the underlying datastore through sharding. These results also showed a reasonable cost in the addition of causal consistency.

Regarding the initial goals of this work, presented in Chapter 1, we have achieved most of them. Our solution provides a good trade-off between data freshness and throughput, having maximized both in a partially replicated scenario (with migrating clients), when comparing to the state of the art solutions. The small size of metadata used in our solution allows it to scale both in the number of datacenters and in the number of machines in each datacenter without any significant performance losses in the handling of metadata. The only objective which was not completely achieved was in the efficiency of partial replication, as our solution still does not employ genuine partial replication, forcing datacenters to handle metadata of operations related to data not replicated in that datacenter.

6.2 Future Work

In this section we present some possible future work regarding the solution presented in this thesis, some of which was thought of in the beginning of this work and some that started showing up as results were analyzed.

Fault Tolerance: Fault tolerance is, of course, an important requirement for every distributed system. In our implementation, the datastore layer is already fault-tolerant, but the causality layer is not. In order to even be considered as a practical solution, this issue needs to be solved. We have thought about several possible techniques to do this: replicating the causality layer, storing the labels to disk when the queues become too long, and requiring each causality node to send acknowledge messages back to the node that delivered a label are some of them.

Causality layer replication: While replicating the causality nodes is a possible way of achieving fault-tolerance, it can also be used in other ways. Decentralizing the processing of labels in each datacenter and dividing it by every replica could be an interesting way to prevent possible bottlenecks in the causality layer, while also possibly allowing more complex metadata in each label to increase concurrency in the execution of operations.

Dynamic datacenters: The implementation created in this work assumes the number of datacenters in the cluster to be static. However, we believe that supporting the dynamic addition and removal of datacenters is something feasible which could be very interesting to implement and useful in practice.

Performance of causality layer: While we are satisfied with the results achieved in this work, we believe that several techniques could be used in the causality layer in order

to improve its performance, thus increasing the overall performance of the system. Some examples of these techniques are the grouping of concurrent operations into a single label, or the delegation of some of the work done in the causality layer to the datastore layer.

Amount of metadata: While the amount of metadata used in this protocol provides satisfactory results, we believe that we could still devise new schemes in order to improve data visibility times (which would also increase overall performance). However, we need to be careful since the exaggerated use of metadata could harm the overall system performance.

Bibliography

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

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

[3] H. Attiya, F. Ellen, and A. Morrison. "Limitations of highly-available eventually-consistent data stores." In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2017), pp. 141–155.

[4] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. "The potential dangers of causal consistency and an explicit solution." In: *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM. 2012, p. 22.

[5] C. Baquero and N. Preguiça. "Why logical clocks are easy." In: *Communications of the ACM* 59.4 (2016), pp. 43–47.

[6] M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. E. Rodrigues. "On the use of Clocks to Enforce Consistency in the Cloud." In: *IEEE Data Eng. Bull.* 38.1 (2015), pp. 18–31.

[7] M. Bravo, L. Rodrigues, and P. V. Roy. "Saturn: a Distributed Metadata Service for Causal Consistency." In: *Proceedings of the 12nd ACM European Conference on Computer Systems*. ACM. 2017, (to appear).

[8] E. Brewer. "CAP twelve years later: How the"rules"have changed." In: *Computer* 45.2 (2012), pp. 23–29.

[9] E. A. Brewer. "Towards robust distributed systems." In: *PODC*. Vol. 7. 2000.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking cloud serving systems with YCSB." In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazon's highly available key-value store." In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[12] P. Dixon. "Shopzilla site redesign: We get what we measure." In: *Velocity Conference Talk*. 2009.

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

[14] 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*. ACM. 2014, p. 3.

[15] M. ETSI. *Mobile Edge Computing-Introductory Technical White Paper*. 2014.

[16] *Examples of read consistency levels in Cassandra*. https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlClientRequestsReadExp.html. Accessed: 2018-03-20.

[17] G. Fettweis, W. Nagel, and W. Lehner. "Pathways to servers of the future: highly adaptive energy efficient computing (haec)." In: *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium. 2012, pp. 1161–1166.

[18] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[19] R. Guerraoui and A. Schiper. "Genuine atomic multicast in asynchronous distributed systems." In: *Theoretical Computer Science* 254.1 (2001), pp. 297–316.

[20] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. "Trade-offs in Replicated Systems." In: *IEEE Data Engineering Bulletin* 39.EPFL-ARTICLE-223701 (2016), pp. 14–26.

[21] C. Gunawardhana, M. Bravo, and L. Rodrigues. "Unobtrusive Deferred Update Stabilization for Efficient Geo-Replication." In: *Proc. of USENIX ATC 17*. Santa Clara, CA: USENIX Association, 2017, pp. 83–95. ISBN: 978-1-931971-38-6.

[22] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX annual technical conference*. Vol. 8. 2010, p. 9.

[23] R. Klophaus. "Riak core: Building distributed applications without shared state." In: *ACM SIGPLAN Commercial Users of Functional Programming*. ACM. 2010, p. 14.

[24] A. Lakshman and P. Malik. "Cassandra: a decentralized structured storage system." In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[25] L. Lamport et al. "Paxos made simple." In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

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

[27] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Stronger Semantics for Low-Latency Geo-Replicated Storage." In: *NSDI*. Vol. 13. 2013, pp. 313–328.

[28] P. Mahajan, L. Alvisi, M. Dahlin, et al. "Consistency, availability, and convergence." In: *University of Texas at Austin Tech Report* 11 (2011).

[29] *Multiple datacenter write requests in Cassandra*. https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlClientRequestsMultiDCWrites.html. Accessed: 2018-03-20.

[30] N. Narkhede, G. Shapira, and T. Palino. *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale*. 1st. O'Reilly Media, Inc., 2017. ISBN: 1491936169, 9781491936160.

[31] R. Rodrigues and P. Druschel. "Peer-to-peer systems." In: *Communications of the ACM* 53.10 (2010), pp. 72–82.

[32] E. Schurman and J. Brutlag. "The user and business impact of server delays, additional bytes, and HTTP chunking in web search." In: *Velocity Web Performance and Operations Conference*. 2009.

[33] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-free replicated data types." In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[34] R. H. Thomas. "A majority consensus approach to concurrency control for multiple copy databases." In: *ACM Transactions on Database Systems (TODS)* 4.2 (1979), pp. 180–209.

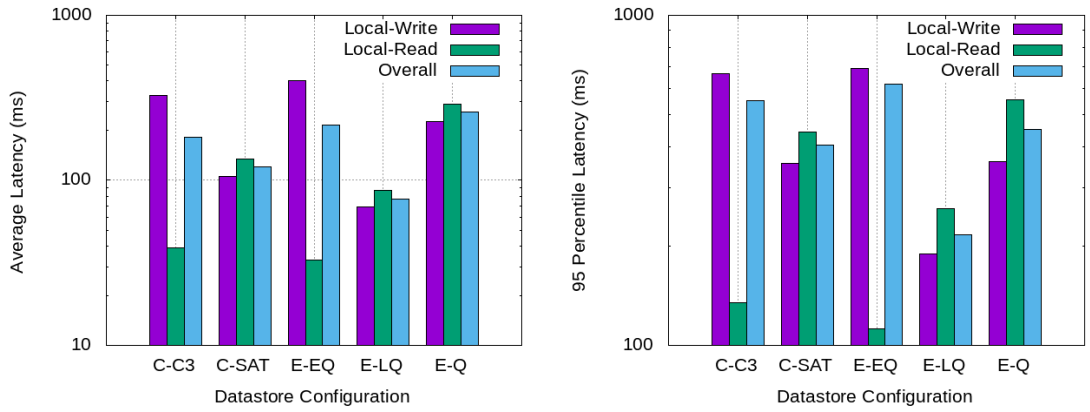
[35] A. Z. Tomsic, T. Crain, and M. Shapiro. "Scaling geo-replicated databases to the MEC environment." In: *Reliable Distributed Systems Workshop (SRDSW), 2015 IEEE 34th Symposium on*. IEEE. 2015, pp. 74–79.

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

ANNEX I

Annex 1 - Extra figures

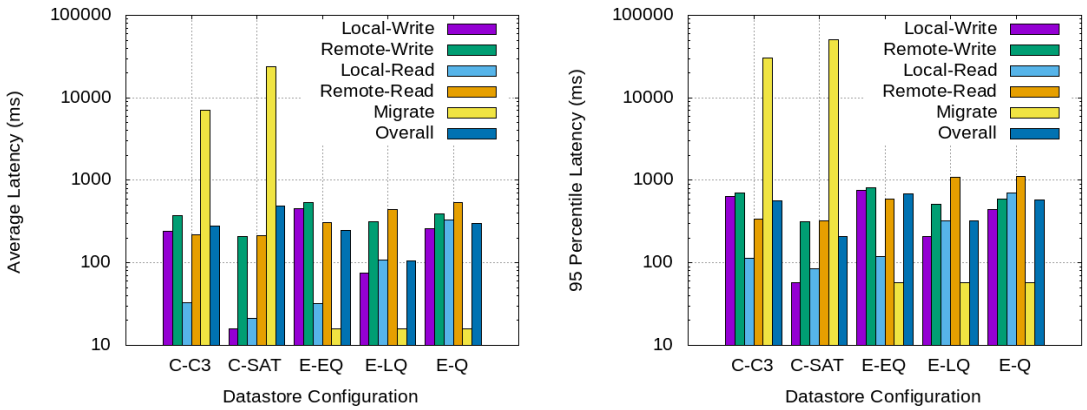
This annex is used to present the extra figures that were left out during the discussion of results in Section 5.6. These figures were left out since we consider them less relevant than the ones presented in that discussion. However, for completeness we provide them here.



a Average latencies

b 95 percentile latencies

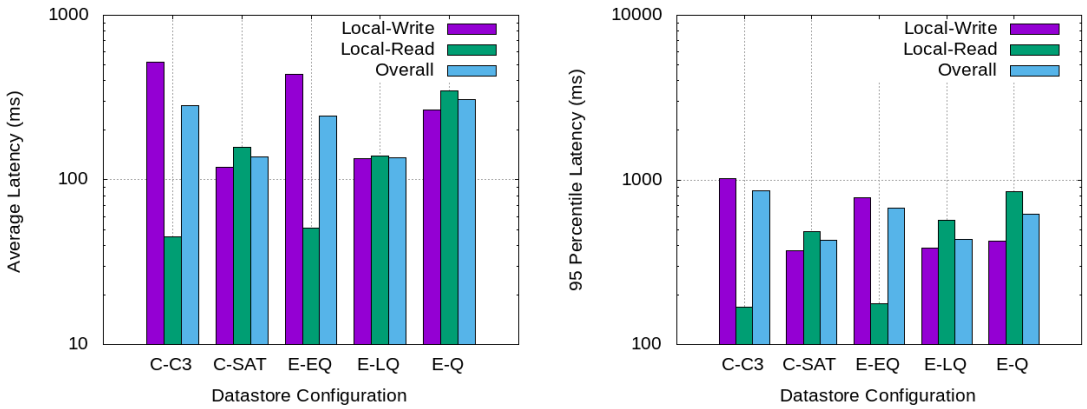
Figure I.1: Latency of each type of operation with 1800 clients and only local operations



a Average latencies

b 95 percentile latencies

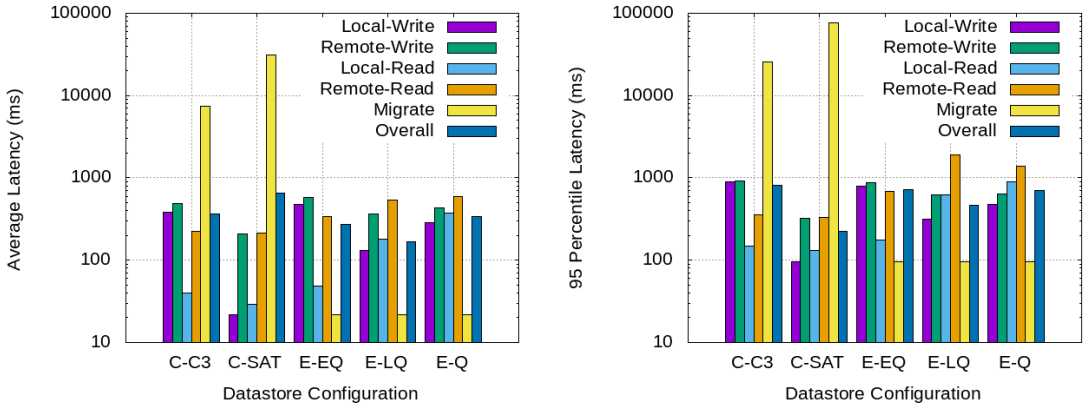
Figure I.2: Latency of each type of operation with 1800 clients and both local and remote operations



a Average latencies

b 95 percentile latencies

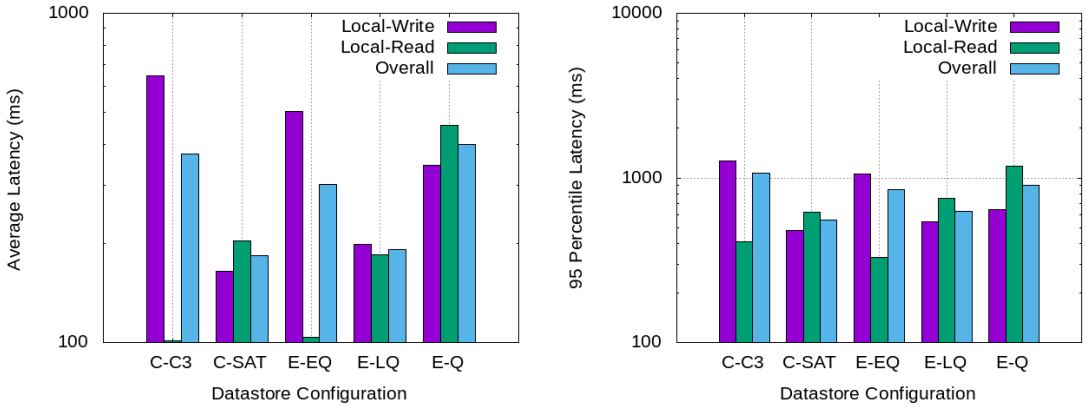
Figure I.3: Latency of each type of operation with 3600 clients and only local operations



a Average latencies

b 95 percentile latencies

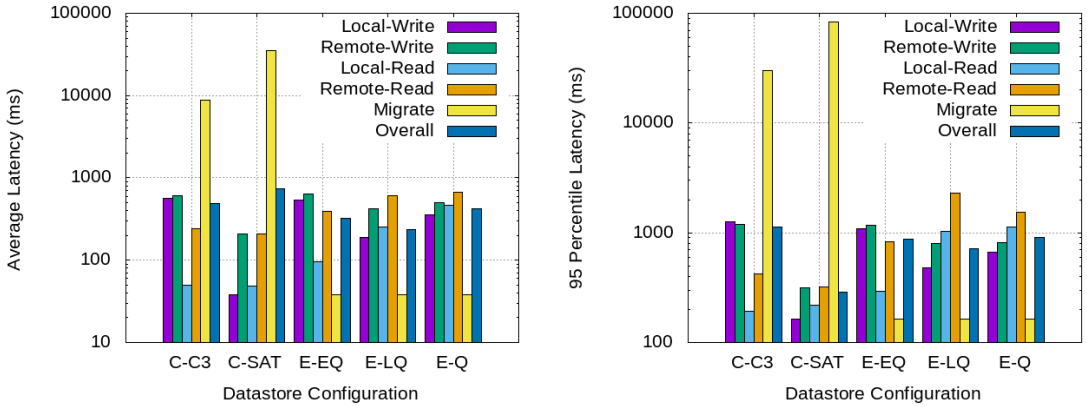
Figure I.4: Latency of each type of operation with 3600 clients and both local and remote operations



a Average latencies

b 95 percentile latencies

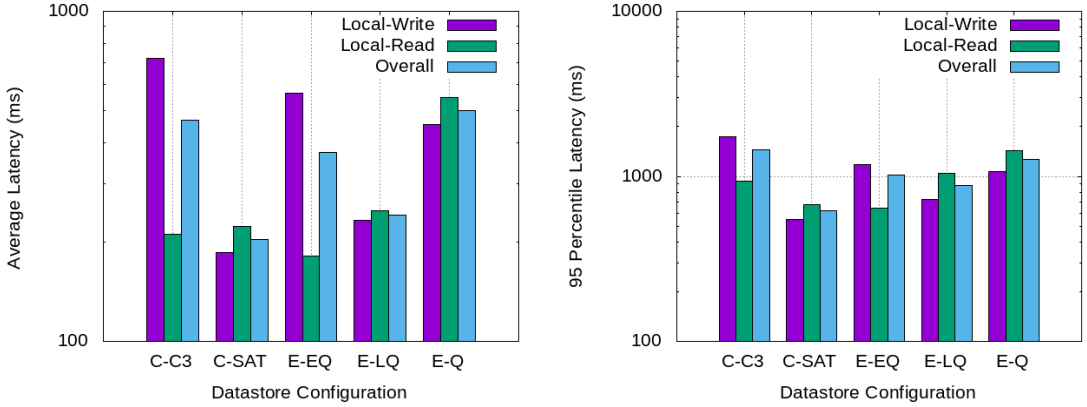
Figure I.5: Latency of each type of operation with 5400 clients and only local operations



a Average latencies

b 95 percentile latencies

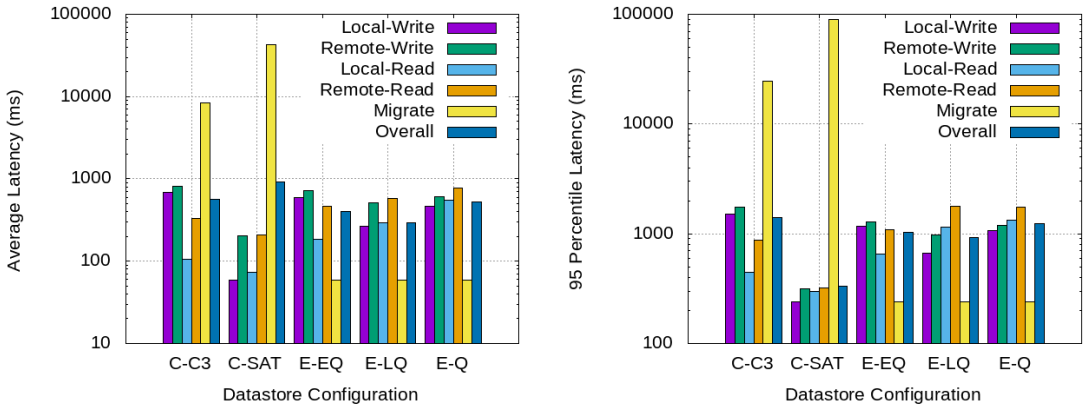
Figure I.6: Latency of each type of operation with 5400 clients and both local and remote operations



a Average latencies

b 95 percentile latencies

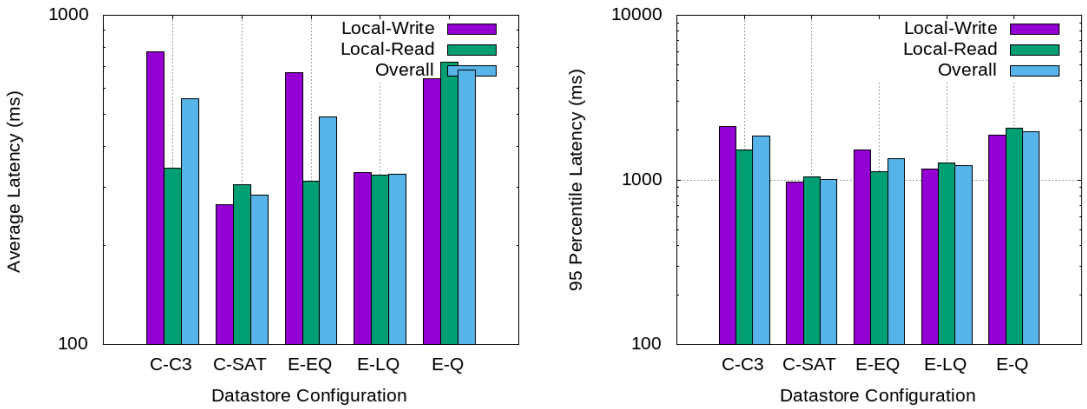
Figure I.7: Latency of each type of operation with 7200 clients and only local operations



a Average latencies

b 95 percentile latencies

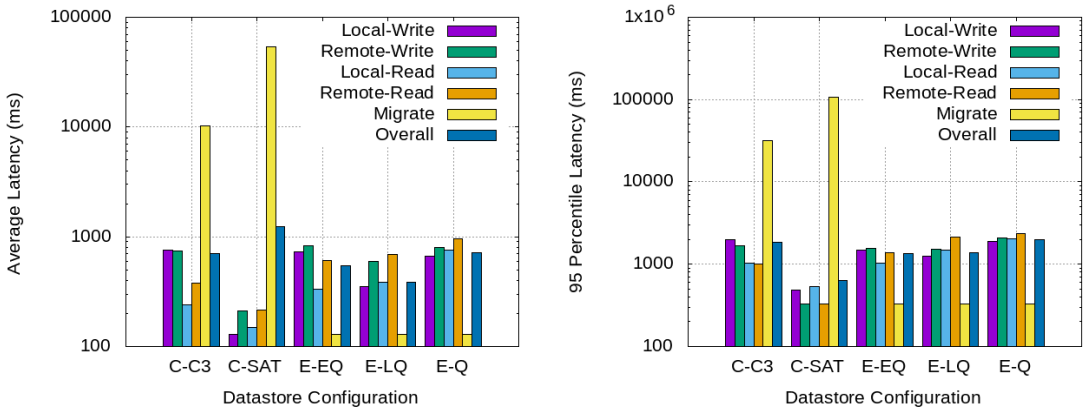
Figure I.8: Latency of each type of operation with 7200 clients and both local and remote operations



a Average latencies

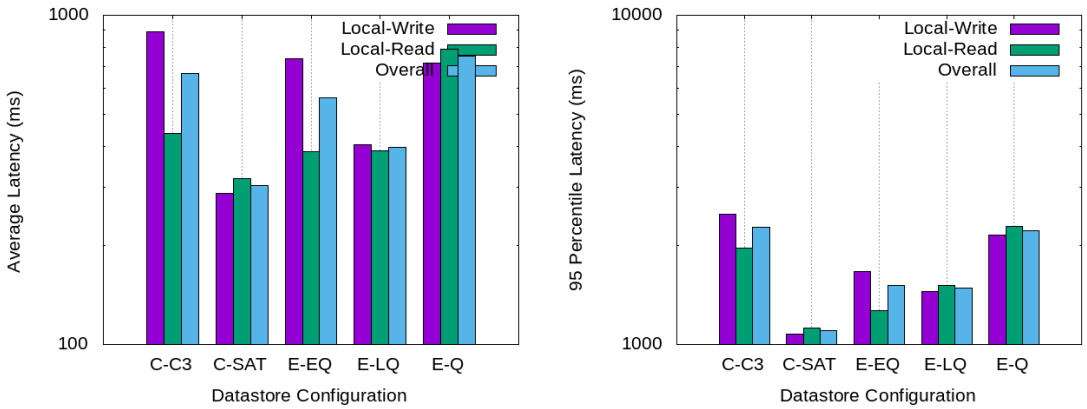
b 95 percentile latencies

Figure I.9: Latency of each type of operation with 10800 clients and only local operations



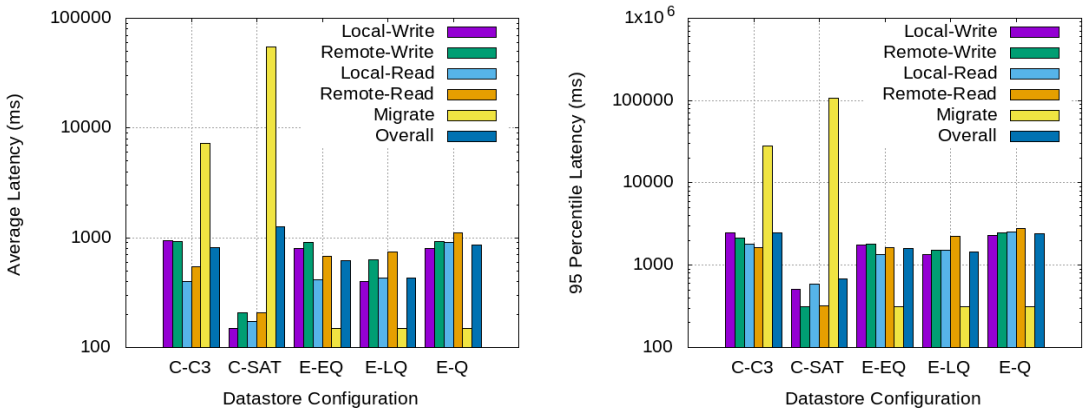
a Average latencies b 95 percentile latencies

Figure I.10: Latency of each type of operation with 10800 clients and both local and remote operations



a Average latencies b 95 percentile latencies

Figure I.11: Latency of each type of operation with 12600 clients and only local operations



a Average latencies

b 95 percentile latencies

Figure I.12: Latency of each type of operation with 12600 clients and both local and remote operations