Pedro Filipe Veiga Fouto

Licenciado em Engenharia Informática

A novel causally consistent datastore with dynamic partial replication

Relatório intermédio para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Nuno Preguiça, Professor Associado, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

Co-orientador: João Leitão, Prof. Auxiliar, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2017

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 leveraging on weaker consistency models.

While system 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, and hence, multiple systems have moved towards providing additional consistency guarantees such as implementing the causal (and causal+) consistency model(s).

In this document, we propose to create a novel geo-replicated data storage system, providing causal+ consistency guarantees and supporting genuine partial replication. This datastore will also tolerate a high amount replicas, potentially located towards the edge of the system, dynamically joining and leaving the system. This system will focus on providing both availability and fault-tolerance with low latency and data freshness to users.

Keywords: Distributed datastore systems, causal+ consistency, geo-replication, genuine 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 disponibiliade, o desenho destes sistemas tem evoluído na tentativa de melhorar estas características, explorando técnicas como a replicação parcial, geo-replicação e usando modelos de consistência mais fracos.

Apesar de existirem sistemas com estas características, normalmente não possuem todas estas propriedades ou fazem-no de forma pouco eficiente, o que resulta em não tirarem total proveito destas. Para além disso, os modelos de consistência fracos (como a consistência eventual) colocam demasiadas responsabilidades nos programadores de aplicações, pelo que muitos sistemas têm tentado proporcionar melhores garantias de consistência, por exemplo implementando o modelo de consistência causal (ou causal+).

Neste documento, propomos a criação de um novo sistema de armazenamento distribuído e geo-replicado, que proporcione as garantias do modelo de consistência causal+ e suporte replicação parcial genuína. Este sistema de armazenamento também deverá tolerar uma grande quantidade de réplicas a entrar e sair do sistema. O sistema será focado em fornecer tanto disponibilidade como tolerância a falhas, mantendo latência baixa e reduzindo atrasos de visibilidade sobre as operaões de escrita sobre os dados.

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Distributed datastores are fundamental infrastructures in most large-scale Internet services. Most of these services, specially recent ones, demand fast response times [11, 22] 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, 30]. In order to provide low latency to end-users, two important properties of the underlying datastore 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 closer to users from diverse locations, thus decreasing response times.

This technique however, can be further improved by using partial replication. In a datastore 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 partial system 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, 16, 33].

The other important property of datastores 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, 27], causal consistency is one of the most attractive weak consistency models, having been implemented in many recent systems [2, 7, 13, 25, 26]. 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.

1.2 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 tradeoff 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, 19]. This tradeoff 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 greater amounts of metadata, sacrificing throughput (and potentially latency). 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 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 geo-replication, they do so inefficiently, not taking full advantage of it. Due to the difficulty of partitioning causal graphs, these systems require partial replicas to handle metadata associated with items that they do not replicate [4, 25, 34]. 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 starts becoming a bottleneck.

Furthermore, most existing solutions assume a datastore static configuration, assuming that changes in the filiation of replicas or datasets replicated by each one are rare, and often triggered by system administrators. Even systems that are not designed to work with dynamic replicas need to be prepared to handle them, since there is also the possibility of a replica to fail requiring it to be replaced, or due to the need to upgrade the system by introducing additional replicas.

As such, replicated systems need to have mechanisms to support replicas joining and leaving without interfering with the system's behavior. To enable better scalability, these mechanisms might be required to be triggered and handled by the system with no human intervention (for instance, to handle workload changes).

1.3 Expected contributions

The work planned, as will further detailed in section 3.2, is based upon the challenges described in the text above. The idea is to create a replication protocol capable of surpassing the existing solutions, while tackling other relevant limitations of existing solutions.

As such, the main expected contribution consists on a replication protocol (and its materialization into a concrete storage system prototype) with the following characteristics:

- Providing efficient causal consistency, with both low metada overhead and low data visibility latency.
- Full support for geo-replication with genuine partial replication, by avoiding the need for replicas to manage metadata concerning items they do not replicate.
- Ability to scale as close to linearly as possible (i.e the system's performance should be proportional to the number of replicas).

As an extension to the previous protocol, we also intend to study the possibility of implementing the following:

- Support for dynamic (and automatic) management of partial replicas (i.e creation and destruction of replicas at will and concurrently). This is particularly challenging to implement while maintaining the previous characteristics.

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 existing consistency models and their characteristics; various techniques used by existing replication protocols; and peer-to-peer networks, which are fundamental solutions associated with the design and implementation of most geo-replicated system.

Chapter 3 describes in more detail the proposed work, including an evaluation and work plan.

Chapter 2

Related Work

In this chapter we discuss relevant related work considering the goals of the work to be conducted in this thesis. In particular we focus on the following topics:
In section 2.1 we study and compare consistency models, with special interest in causal consistency.
In section 2.2 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 2.3 peer-to-peer network are discussed, since they're the base of every replicated system.

2.1 Consistency models

According to the original CAP theorem [9, 17], 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 web systems where user experience is essential to ensure success, as seen in the NoSQL movement for example, choose availability over strong consistency [2, 11, 22].

2.1.1 Strong Consistency

A system that chooses consistency over availability typically focuses on providing guarantees in line with on of the existing strong consistency models. In these models every operation is observed by all users in the same order, meaning 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:

Sequential Consistency: For a system to provide sequential consistency, 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 sequential consistency. 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 sequential consistency 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.

2.1.2 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"[32] approach being the most common, although the use of CRDTs [31] has gained some popularity recently [1, 21].

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.

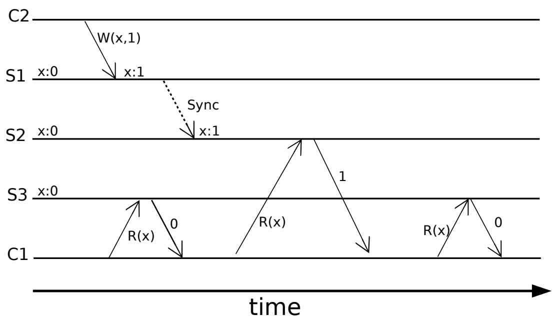
This consistency model requires the system to keep track of causal dependencies between operations and ensures that those operations are always seen by clients in an order that respects their causality relations.

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.

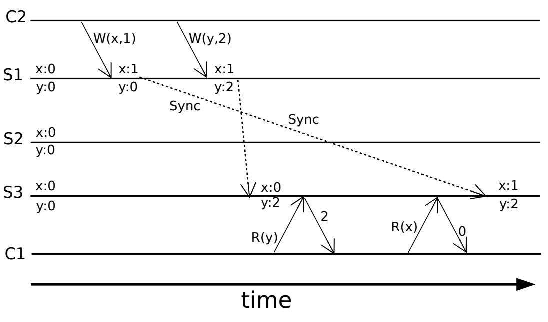
Operations that don't have any causal dependency are called concurrent operations, since they don't have any relation between them being fully independent. Concurrent operations do not have to be presented to users in any particular order because of this. Simultaneous (and therefore concurrent) writes, 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.

2.1.2.1 Differences between eventual and causal consistency

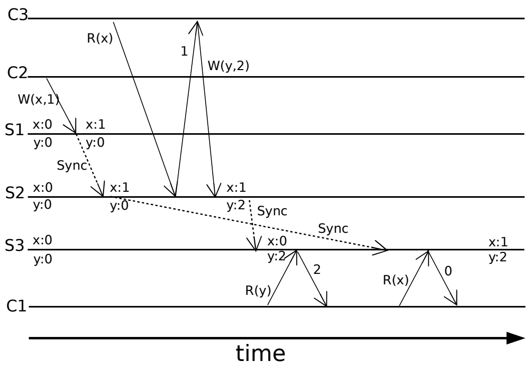
In this section, we present examples on the differences between eventual and causal consistency, using figures in 2.1 as reference. In these figures, *c1*, *c2* and *c3* refer to clients operating in a system consisting on three replicas: *s1*, *s2*, *s3*. *x* and *y* represent the keys of objects stored 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



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

parameter in a read operation in 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 2.1c, 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 2.1b 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.

2.2 Replication Protocols

Many distributed systems resort to some sort of data replication. Replicating data to 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 operation correctly. 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.

2.2.1 Replica Location

Co-located Co-locating replicas consist of having a distributed system replicated in 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 means 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 means replicas are closer to the users, thus decreasing latency in communications between the user and a replica. Since these systems usually focus on user experience, this trade-off is acceptable (and often welcomed).

2.2.2 Replications Schemes

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 the 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 user experience, while avoiding the need to deploy a full replica (which would incur in higher costs).

Genuine Partial Replication[18]: 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 data that is 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 modifies less frequently.

2.2.3 Update Propagation/Synchronization

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 the 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 time. 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 eventually 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 allow 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 time. 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).

2.2.4 Multimaster / Primary backup

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 updated).

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 to 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 [23], or other coordination systems such as Zookeeper [20].

2.2.5 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.

2.2.5.1 Causal history

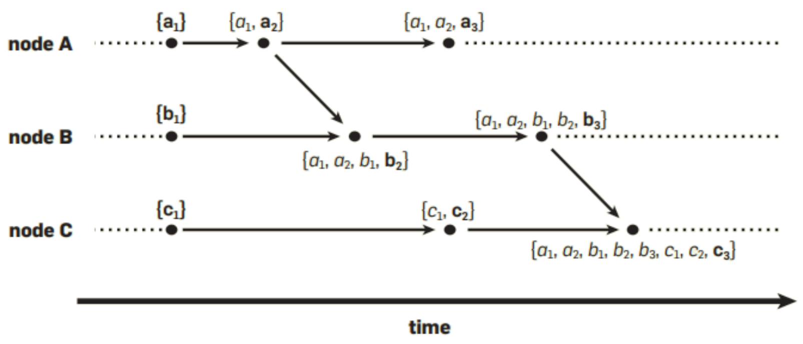


Figure 2.2: An example causal history - Adapted from [5]

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 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 highly 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.

Which 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: $\forall \text{i} : V_z[\text{i}] = \mathbf{max}(V_x[\text{i}], V_y[\text{i}])$

- To check if there's a causal dependency between two events, checking if every position of the vector identifying an event is greater or equals than 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 is 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 [25]: 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.

2.2.5.2 Metadata Propagation

A different way to guarantee causality is to control the propagation of meta-data 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. Data centers 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.

2.2.6 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).

2.2.7 Conflict Resolution

When employing weak consistency models, it is predictable that two replicas might at some point evolve to divergent states, for instance, if two users execute concurrent updates over the same data item in two different replicas. These conflicting states must eventually be resolved by the system into a single merged state, so that the system can remain consistent. This is usually done in two steps: first the system needs to exchange the state of each replica between replicas (anti-entropy); then an appropriate final state must be chosen, based on the state of each diverging replica (reconciliation).

There are several techniques to resolve these kinds of conflicts, some of the most common being:

Last Writer Wins: This is the most trivial conflict resolution technique. When there are concurrent updates, we simply choose the most recent one (based on the system's clock or some deterministic ordering criteria) and discard all the others. While trivial to implement on a single server, when using multiple replicas clocks may be out of sync, leading to incorrect decisions. Even if the right decision is made, there are updates that are discarded anyway, which means this technique might not be appropriate for every system, as there might be lost updates.

For instance, if we imagine two users are using some service to shop online, each connected to a different server, and both add a different item to the shopping cart concurrently, each server will see the shopping cart in a different state: one might see, for example, items A,B,C and the other A,B,D. In this situation, when converging states, either item C or D will be lost. Some systems choose to live with this possibility, resolving it later in some other way (sometimes manually).

Programatic Merge: This technique is based on letting the programmer decide what to do with conflicts. Usually this is done by either giving replicas some kind of merge procedure, which helps them decide how to merge diverging states, or by requiring replicas to expose the diverging states to the client application, which then reconciles and writes back the new converged state. By being given control over how the merge procedure operates, the programmer can chose the best way to handle conflicts depending on the application goals.

For instance, using the same example as above, a better merge procedure would be the union of both carts, resulting in the final cart containing all four items (A, B, C, and D), which means none of the updates was discarded. In some cases however, this technique might force the programmer to deal with extremely large and complex possibilities, which might lead to errors in the programmed solution.

Commutative operations: Another technique is to design the system to only allow commutative operations to be executed. As the name suggests, independently from the order in which these operations are executed, the final result is always the same (as long as every replica receives and applies every operation).

A simple example is a system consisting of a counter, in which the only operations are increment and decrement, in this case, if every replica sees every operation, the final result is always the same independently of the ordering which they are applied. By using only commutative operations, conflict resolution becomes a matter of simply ensuring that every replica executes every update, however, since most systems are more complex than a simple counter, designing commutative operations is not trivial, and in some cases might be impossible.

Commonly used commutative operation techniques are:

Operational transformation (OT): The idea of OT is to transform parameters of operations, depending on the effects of previous concurrent operations, so that the result is always consistent. This technique was originally designed to guarantee consistency and handle concurrency in collaborative editing applications.

For instance, in a collaborative text document containing *"abc"* being edited by two users, one user inserts the character x on position 0 and the other removes the character c from position 2, concurrently. If both execute their operation before receiving the other, their states will diverge, one of them will see *"xac"* (the wrong result) while the other will see *"xab"* (the correct result). For the user who issued the insert operation to get the correct result, the remove operation needs to be transformed, from *"remove c from position 2"* to *"remove c from position 3"*.

This technique has been studied extensively and has evolved considerably. However, while the OT approach seems to work naturally with text documents, to use it on other types of systems, with more complex interfaces and data types, would not be so trivial. It was also demonstrated that most OT algorithms proposed for a decentralized OT architecture are incorrect [28].

CRDTs: Conflict-free Replicated Data Types are replicated data types that guarantee strong eventual consistency while being very scalable and fault-tolerant. Since conflicts are impossible in these data types, no strong synchronization is required to ensure convergence. This means updates can immediately execute locally, unaffected by network latency or faults, and later be propagated to the other replicas.

There are two types of CRDTs:

State Based or convergent replicated data types (CvRDTs). When using CvRDTs, the complete set of possible states should form a semilattice. To execute the merge operation it's only needed to calculate the least upper bound in the semilattice, which corresponds to the resulting state. This means that, when merging states, all replicas will eventually converge to the same state. CvRDTs only require communication primitives with guarantees of eventual delivery between replicas, however sending the entire state might be inefficient for large objects. Gossip protocols are particularly useful when dealing with CvRDTs.

Operation Based or commutative replicated data types (CmRDTs). In this type of CRDT concurrent operations can be executed locally and then propagated to the other replicas, since concurrent operations are commutative. However there needs to be a reliable broadcast communication channel with ordering guarantees (such as causal order) for propagating operations between replicas, to avoid replicas from evolving to states that cannot be converged.

In summary, both classes of CRDTs are guaranteed to eventually converge towards a common final state, the main difference being what information is propagated between replicas. CvRDTs propagate the entire replica state, while CmRDTs only need to propagate the operations they receive with some ordering guarantees.

2.2.8 Existing systems

COPS[25] was the first system to introduce the concept of causal+, 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 before making their results visible. It uses client-side metadata to keep track of the dependencies for each client operation.

Eiger[26] 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 [22]), 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 to enforce causality with.

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. This

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 is essential to ensure scalability in a system that supports partial replication. 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).

2.3 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[29].

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's 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.

2.3.1 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 its 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.

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).

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

Chapter 3

Proposed Work

This chapter begins by describing the assumed system model which will be the basis for conducting the work proposed in this document. In the second section, the intuition for the proposed solution itself is presented, in a progressive manner. Finally, in the last two section, the evaluation and work plans are described.

3.1 System Model

In this section, we present the assumptions for the system model of our replicated data storage system. In this system model, we consider two types of replicas, classified by their lifetime: stable replicas and ephemeral replicas.

Stable replicas: Stable replicas are characterized by their long lifetime, usually staying in the system for long periods of time (possibly the entire lifetime of the system). Since these replicas are always connected to the system, they contribute very little to the system's dynamism. Stable replicas can either replicate the entire set of data of a system or just a subset, however when replicating only a subset of data, that subset is usually a considerable percentage of the entire dataset. These replicas are usually machines with considerable amounts of resources, such as machines in data centers or dedicated servers.

Ephemeral replicas: As the name suggests, the most particular property of ephemeral replicas is their short lifetime, which can be as low as a few minutes. Due to this characteristic, they usually contribute highly to the system's dynamism. As a consequence, a system that supports ephemeral replicas must be prepared to tolerate an high amount of nodes entering and leaving it. Ephemeral replicas can consist of very different types of physical devices, ranging from data centers all the way to, in the limit, user devices (such as laptops or smartphones). This heterogeneity means not every ephemeral replica behaves the same way, having different characteristics, such as:

Capacity: While a data center can be expected to hold a large portion of data, the same cannot be said about, for instance, a laptop. As such there is a limit in how much data an ephemeral replica can replicate

Lifespan: While some replicas can stay connected to the system for days or weeks, other replicas may stay for very short periods (minutes).

Processing Power and Bandwidth: While some replicas with more resources might be able to easily maintain large amounts of data and send and receive events at a large rate, other replicas might have limited processing power and/or bandwidth which limits their capability to do so.

Interests: While some replicas will maintain the same data during their lifetime, others might have more inconstant interests, and change their replicated dataset multiple times.

3.2 Proposed Solution

In this section, we outline our proposed solution for creating a geo-replicated data storage system, with causal+ consistency guarantees. Our system should support genuine partial replication while at the same time tolerating a high amount of dynamic partial replicas entering and leaving the system concurrently.

3.2.1 Static Solution

To address this challenging problem, we will first consider a limited system model, where only stable replicas exist. This means the system will not need to support replicas dynamically entering and leaving (at this stage).

As a starting point, we will use a solution based on Saturn, which we consider to be the state of the art in the context of this work. While being the closest system to what we're trying to achieve, it still does not solve a several problems that we're tackling, such as:

Fault Tolerance: As explained in 2.2.8, in Saturn metadata is delivered to data centers using a tree structure, with the data centers being the leaves of that tree. This means there is only one path to propagate metadata between each pair of data centers. In case of a failure or partition in a node of the tree, data centers may stop receiving metadata until a new tree is generated (which is an expensive operation). Saturn uses a fall-back technique consisting in sending timestamps directly with data updates, which maintains the system causally consistent even if the metadata service fails. However this fall-back mechanism is much less efficient, and potentially breaks the properties of genuine replication.

As a solution for this problem, we intend to explore two possible solutions: using a more efficient tree structure, capable of efficiently reconfiguring itself in case of a failure or, alternatively; using a different structure to connect replicas, such as a cyclic graph, which would have the advantage of having redundant connections, however causality tracking might be trickier to achieve, using such a solution.

Latency and congestion: While the simplest structure that allows the tracking of causal dependencies is a tree, another problem arises: metadata must always travel in a FIFO order between nodes of the tree, with a single path between two nodes. Depending on the number of nodes in the tree, this path may pass through a large number of intermediate nodes, increasing latency in the system and possibly congesting paths.

Going back to the fault tolerance problem, we proposed two solutions: a more efficient tree and using a cyclic graph instead. While a more efficient tree probably will not address these challenges in an effective way, a graph might. Using a graph to propagate metadata can both decrease latency and avoid congesting due to the possibility to introduce redundant paths between nodes.

Tracking causality: After considering the previous challenges, using a graph instead of a tree seems to be the most promising solution. However, with the use of a graph comes a new problem: causality tracking. In a tree structure this can be achieved simply by propagating data between nodes in an ordered fashion, guaranteeing that an operation is always propagated to a node after all the operations it depends on. However, in a cyclic graph, with redundant paths, this is not so simple.

A possible solution for this challenge would be to transfer logs (as opposed to transferring operations) between nodes. This protocol would need to merge the logs received in each node with their local logs, maintaining the same relative orders of operations. We would also need to guarantee that every log is propagated across every path in the graph. While this currently seems to be the most appropriate solution, we will still further study this and other possible solutions for this problem.

Dynamism: Even though we're still only considering stable replicas, the system still needs to support some dynamism, as replicas may fail and need to be replaced, or replicas might need to change the set of data they replicate. As such, to handle these situations, the chosen structure needs to be able to efficiently recon itself, without impairing system performance.

While the final structure to propagate causality tracking metadata of the system is not yet defined, the cyclic graph seems to be the most promising solution for this system model. As such, for the next subsection we will assume that we have a working solution, consisting of the replicas of the system being connected by a cyclic graph, with a communication protocol for propagating causality tracking metadata capable of guaranteeing causal consistency.

3.2.2 Dynamic extension

After achieving a satisfactory solution for the previous system model, we intend to progressively remove restrictions from the system model until we address all challenges described in the beginning of this section. In this stage we will start considering the possibility of ephemeral replicas entering the system.

3.2.2.1 Full replica assumption

Before considering the full extent of our system model, in this stage we still maintain one assumption: for every ephemeral replica that enters the system, there is at least one stable replica that maintains a superset of the data that any ephemeral replica might try to replicate. This means that an ephemeral replica can always find a stable replica with all the data it is interested in.

In order to support this new type of replica, we consider the use of multiple tree structures, with each tree having as root a static replica (remember that all static replicas would be connected through a cyclic graph). This solution behaves the following way:

- When a new ephemeral replicas enters the system, it looks for the best existing tree to join, with the following conditions:
  - The parent replica must have a superset of the data the new replicas wishes to replicate, and should preferentially be the replica with the smallest superset.
  - The parent replica should be the geographically closest replica to the new replica that complies with the previous condition, in order to minimize latency.
  - If no such parent replica exists in any tree, the new replica creates a new tree with a static node as its parent node.

  A possible way to implement this join protocol is to have the new replica find the closest static replica that has a superset of the data the new replica is interested in and then iterate that replica's tree looking for the best replica to use as parent.

- When an ephemeral replica leaves the system, there needs to be some mechanism to prevent tree branches for breaking (for instance, if the replica that left was the parent of some other replica, the now orphan replica needs to find a new parent replica). In order to deal with this, several solutions can be considered:
  - Each replica can store the path to the root of its tree. This means an orphan replica can find a new parent replica very quickly, however this would also

  mean that every time a replica joins or leaves a tree, other replicas in that tree needs to be notified to update their view of the tree topology.

  – Every time a replica leaves the system, replicas that used that replica as parent re-execute the join protocol, looking for a new parent node in the entire system. This would mean slower recovery, but would avoid the need to store and propagate information about changes in the tree structure.

  – Using a mechanism based on the one presented in Plumtree [24], that uses a gossip protocol to improve the recovery time for handling faults that partition the tree.

3.2.2.2 No full replicas

After having a satisfactory solution to the previous system model, we will now consider the complete system model described in 3.1, without any restriction. This means that we now consider the possibility for a new ephemeral replica to enter the system without there being any replica already in the system that maintains a superset of the data the new replica is interested in. This means that using the tree structure described above in which an ephemeral replica is connected to a single parent replica will no longer work.

In order to solve this new challenge, we will consider two possible solutions to support this particular case:

- After concluding there is no possible parent replica in any tree with a superset of the data it is interested in, the new replica can join the cyclic graph composed by the static replicas. This means that the graph needs a mechanism to easily allow replicas to join and leave it.

- After concluding there is no adequate parent replica, the new replica can choose multiple replicas from a tree to act as its parents. This would mean that the replica could receive duplicated data. A possible solution for this would be to make only one of the parents propagate metadata (even if it doesn't replicate the data for that metadata).

3.3 Evaluation

We plan to evaluate the proposed solution in the following way:

- Regarding the initial static solution, we will compare our system with existing state-of-the-art solutions with similar characteristics (i.e partial geo-replication and causal consistency model). In this phase, we are specially interested in measuring throughput (the number of operations executed per unit of time) and update visibility times.

- For the dynamic extension, we plan to study the overhead induced by implementing support for ephemeral replicas. We are interested in seeing how the system behaves with an high amount of replicas entering or leaving the system concurrently, and how long it takes to recover from failures.
- A possible way way to conduct the evaluation is by using a cloud infrastructure (such as Amazon EC2 [1]), resorting to a workload generator (such as Yahoo Cloud Serving Benchmark [10]). For the dynamic replica behavior, we will model multiple expected behaviors considering particularly relevant use cases.

3.4 Work Plan

In this section we present the work plan for the elaboration of this work. We start by dividing the work plan in three phases, corresponding to the phases described in the previous section. In each phase we will design, implement, test and evaluate the achieved solution before passing to the next one. After completing the the last phase we will evaluated more extensively the solution, and finish by writing the dissertation. This work plan is to be completed is around seven months. Table 3.1 details the timespan for each task and figure 3.1 shows a Gantt graph with the proposed schedule.

Table 3.1: Schedule

Task	Start Date	End Date	Weeks
Static Solution	20 February	21 April	9
Design			
Implementation			
Testing and Evaluation			
Dynamic Extension: Full Replica Assumption	24 April	9 June	7
Design			
Implementation			
Testing and Evaluation			
Dynamic Extension: No Full Replicas	12 June	14 July	5
Design			
Implementation			
Testing and Evaluation			
Final Evaluation	17 July	11 August	4
Writing	14 August	22 September	6

Static Solution and Dynamic Extension (Full Replica Assumption and No Full Replica)
Each of the first three phases will have a similar work plan. We will start by choosing a system design capable of providing the necessary characteristics for the system

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

model corresponding to this phase. After choosing an adequate design, we will start implementing it. This implementation is progressive, meaning the work done in later tasks are to be built on top of the work done in the previous phase. At the end of each phase, some testing and preliminary evaluation will be conducted to determine if the implementation is in a satisfactory state (and to validate the solution) in order to move on to the next phase.

Article Submission We will try to submit a preliminary version of this work to the Portuguese national informatics conference (INFORUM) by end of the work on the dynamic extension assuming the availability of full replication, the second main task in the presented schedule (table 3.1).

Final Evaluation After completing the system's implementation, a more extended and thorough evaluation will be done in order to understand how does this system compare to other existing systems.

Writing This task will consist in writing the dissertation, marking the end of this work.

3.5 Summary

In this chapter we have discussed the system model in which this work will be based on. We also discussed the intuition associated with the solution that we are going to explore to face all challenges discussed previously in this document. The evaluation plan that will serve to validate our solution was presented and the chapter concludes with a brief presentation of the work plan to serve as guideline during the remainder of the work leading to the writing of the thesis.

	February				March				April					May				June				July					August				September			
	5	12	19	26	5	12	19	26	2	9	16	23	30	7	14	21	28	4	11	18	25	2	9	16	23	30	6	13	20	27	3	10	17	24
Static Solution																																		
Design			■	■	■																													
Implementation					■	■	■	■	■																									
Testing and Evaluation									■	■	■																							
Dynamic Extension: Full Replica Assumption																																		
Design												■	■	■																				
Implementation														■	■	■	■																	
Testing and Evaluation																■	■	■																
Dynamic Extension: No Full Replica																																		
Design																			■	■														
Implementation																				■	■	■												
Testing and Evaluation																						■	■											
Final Evaluation																								■	■	■	■							
Writing																												■	■	■	■	■	■	

Figure 3.1: Gantt graph with proposed work schedule

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] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

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

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

[24] J. Leitao, J. Pereira, and L. Rodrigues. “Epidemic broadcast trees.” In: *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*. IEEE. 2007, pp. 301–310.

[25] 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.

[26] 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.

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

[28] G. Oster, P. Urso, P. Molli, and A. Imine. “Proving correctness of transformation functions in collaborative editing systems.” Doctoral dissertation. INRIA, 2005.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.