Variable Consistency Messaging Layer

José Henrique Sobral Santos

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. João Coelho Garcia

Examination Committee

Chairperson: Prof. Francisco João Duarte Cordeiro Correia dos Santos
Supervisor: Prof. João Coelho Garcia
Member of the Committee: Prof. João Pedro Faria Mendonça Barreto

September 2020

Acknowledgements

I would like to express my gratitude to everyone who contributed in any way to complete this work.

I would first like to thank my supervisor João Garcia for all the effort, patience, time and discussions that helped me to rethink and improve this work, but also I have to thank all the positive pressure he put on me to complete it.

I would like to give a special thanks to João Loff who, without any obligation to do so, helped me in everything he could from the first moment. His knowledge and constant feedback undoubtedly contributed to a better and more complete work.

Last but not least, I would also like to thank my family and friends who somehow helped me throughout this thesis.

I couldn't finish without showing my gratitude to my mother, who always gave me everything I needed and made it possible for me to study where and what I wanted.

Abstract

Geo-distributed systems provide high availability, low-latency, and fault tolerance through replication to different locations. The major downside is that replication can lead to divergences between replicas, either caused by network failures or simply by a network delay. Handling these divergences is usually left to a consistency protocol which is implemented by the underlying system. Nowadays, systems tend to implement a single consistency model embedded in their implementation. When the system requirements change and the consistency model is no longer appropriated, developers are left with one of two choices: either (i) the system needs to be deeply rewritten or (ii) replaced by a different system, with a new set of consistency guarantees.

We propose a framework that abstracts the implementation of the consistency model, into a set of well-defined modules. This structural abstraction aims to frame the most common consistency protocols within these modules, as well as to ease the switching of consistency protocol in the targeted system. We have evaluated our framework by measuring the throughput and overhead between the original and our modified implementation with the framework of two different storage systems. The measurements show that this modularity and abstraction have an associated overhead. However, it is compensated by the flexibility and ease in changing modules and the respective consistency model offered.

Keywords: Consistency, Modularity, Replication, Framework, Distributed Systems

Resumo

Os sistemas geo-distribuídos providenciam alta disponibilidade, baixa latência e tolerância a falhas através da replicação em diferentes localizações. A principal desvantagem é que a replicação pode levar a estados de divergência entre as réplicas, causados por falhas de comunicação ou simplesmente devido a atrasos na rede. O tratamento dessas divergências é geralmente deixado a um protocolo de consistência que é implementado pelo sistema. Hoje em dia, os sistemas tendem a implementar um único modelo de consistência que se encontra embutido na sua implentação. Quando os requisitos do sistema mudam e o modelo de consistência precisa de ser ajustado, os programadores ficam com uma de duas opções: ou (i) o código do sistema é profundamente reescrito ou (ii) o sistema é substituído por um diferente que oferece um novo conjunto de garantias de consistência.

Neste documento, propomos uma *framework* que abstrai a implementação do modelo de consistência para um conjunto de módulos bem definidos. Esta abstração estrutural visa enquadrar os protocolos de consistência mais comuns dentro desses módulos, bem como facilitar a troca de protocolo de consistência no sistema. Avaliámos a nossa *framework* medindo a taxa de transferência e o custo adicional entre a implementação original e a modificada com a *framework* de dois sistemas de armazenamento diferentes. As medições mostram que essa modularidade e abstração têm uma penalidade de desempenho associada. No entanto, esta é compensada pela flexibilidade e facilidade de troca de módulos e respectivo modelo de consistência oferecida.

Palavras-chave: Consistência, Modularidade, Replicação, Framework, Sistemas Distribuídos

Contents

List of Figures **xiii**

List of Tables **xv**

Listings **xvii**

Acronyms **xix**

1 Introduction **1**
1.1 Contributions 2
1.2 Thesis Outline 3

2 Related Work **5**
2.1 Replication 5
2.1.1 Active Replication 6
2.1.2 Passive Replication 6
2.1.3 Lazy Replication 6
2.1.4 Full Replication 6
2.1.5 Partial Replication 7
2.2 CAP Theorem 7
2.3 ALPS 8
2.4 Consistency 8
2.4.1 Linearizability 10
2.4.2 Sequential Consistency 11
2.4.3 Per-Record Sequential Consistency 11
2.4.4 Causal Consistency 11
2.4.5 Session Guarantees 12
2.4.6 Eventual Consistency 13
2.5 Existing Implementations 13
2.5.1 Dynamo: Amazon's Highly Available Key-value Store 13
2.5.2 PNUTS: Yahoo!'s Hosted Data Serving Platform 14
2.5.3 Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS 15

2.5.4 Spanner: Google's Globally-Distributed Database 16
2.5.5 Bolt-On Causal Consistency 17
2.5.6 Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary 18
2.6 Discussion of Existing Implementations 19

3 Architecture **21**
3.1 Group Membership 22
3.2 Ordering 22
3.3 Replication 23
3.4 Delivery Condition 23
3.5 Quorum 24
3.6 Communication 24
3.6.1 Internal communication API 25
3.6.2 External communication API 25
3.7 Framework API 25
3.8 Inter-module interactions 26

4 Implementation **29**
4.1 Methodology 29
4.2 Programming Language Choice 29
4.3 Choice of replicated storage systems 30
4.3.1 DKVF 30
4.3.2 Project Voldemort 30
4.3.3 Discussion 31
4.4 Code Structure 31
4.5 Main Class 32
4.5.1 Switching modules and versioning 33
4.6 Implementation experience 34

5 Evaluation **35**
5.1 Methodology 35
5.2 DKVF 35
5.2.1 Experimental Setup 35
5.2.2 Experimental Results 36
5.3 Project Voldemort 37
5.3.1 Experimental Setup 37
5.3.2 Experimental Results 38
5.4 Discussion 39

6 Conclusion **41**
6.1 Future Work 41

Bibliography **43**

List of Figures

2.1 Example of an inconsistent system 9
2.2 Consistency models tree, adapted from [35] 10
2.3 COPS architecture (from [25]) 15
2.4 Spanner Architecture (from [45]) 16
2.5 Spannerserver Software stack (from [45]) 17
2.6 Bolt-on architecture: a causally consistent shim layer mediates access to an underlying eventually consistent data store [48] 18
2.7 Causal Cuts [48] 18

3.1 New message flow 26
3.2 Replication Message Flow 28

4.1 Framework code structure 32

5.1 Experimental cluster representation 36

List of Tables

5.1 COPS with DKVF original implementation - 50:50 operations ratio 36
5.2 Modified COPS with DKVF - 50:50 operations ratio 36
5.3 COPS with DKVF original vs modified overhead - 50:50 operations ratio . . . 37
5.4 COPS with DKVF original implementation - 95:05 operations ratio 37
5.5 Modified COPS with DKVF - 95:05 operations ratio 37
5.6 COPS with DKVF original vs modified overhead - 95:05 operations ratio . . . 37
5.7 Project Voldemort - 50:50 operations ratio 38
5.8 Project Voldemort - 95:05 operations ratio 38

Listings

4.1 New put message on framework 33

Acronyms

ALP	availability, low-latency, partition-tolerance
ALPS	availability, low-latency, partition-tolerance, scalability
ATM	automated teller machine
CAP	consistency, availability, partition-tolerance
COPS	clusters of order-preserving servers
CPU	central process unit
DKVF	distributed key-value framework
IP	internet protocol
RAM	random access memory
YCSB	yahoo! cloud serving benchmark

Chapter 1

Introduction

The growth of the Internet has changed not only the way we see the world but also the way engineers design and develop computer systems. In the 20th century, applications were developed with all parts integrated in the application itself, e.g., a web server application included the web server itself, some kind of data storage and it was all compiled and run directly on a single server [1]. When a system needed to be upgraded to provide better performance or greater capacity, the vertical scaling strategy solved the scaling needs by buying a better CPU or adding bigger and better hard disks or RAM's. This approach worked for the reality of those times. However, nowadays, for example, Google Search receives more than 3.5 billion search queries every day [2], making it almost impossible to have a single server in the world able to handle this amount of processing.

Given the growth rate of the Internet and the seeming end of Moore's law [3], solutions that used a vertical scaling were deemed obsolete and unsustainable [1]. The alternative was to change the focus from vertical scaling to horizontal scaling. The focus was no longer on making a single machine better but adding more machines to a pool of resources. The earliest horizontal scaling was just running duplicates of the web server [1], but nowadays, pursuant all the advancements of cloud technology, microservices architecture has emerged [4].

Taking advantage of horizontal scaling, a huge application can be split into smaller services that can still perform a meaningful task [5]. Applying a microservice architecture to the previous web server example, instead of having a single web server running all the requests, the application is split up in services, such as: user authentication, database model service, and so on. The decentralized governance of the services allows services to be anywhere across servers and replicated as needed, instead of creating clone instances of the entire application every time.

This new architectural model brought new challenges, such as coordination and consistency between nodes spread all over the world, dynamic group membership and availability of the entire system. To guarantee availability, a system should be replicated across different machines [6], keeping the system accessible and operational even in cases of catastrophes. Thus, this raises a problem of consistency between replicas. When a system is being replicated, due to the network connections, the order in which each replica receives the messages may be different or even never receive one of the messages. Therefore, the replicas could be in different states, diverging between them.

When someone is building a new system, there are a lot of decisions to be made about the system design, especially about system guarantees. Choosing a consistency model, which is a set of rules for visibility and apparent order of updates to the system's objects [7], is a problem itself. If, on the one hand, making the right decision about what model should be used can be really difficult [8], on the other, the business requirements that supported that decision may change and the consistency model chosen becomes no longer appropriate. Let's imagine a company whose core replicated storage system was initially built with eventual consistency guarantees. The company grows and businesses evolve, changing the initial requirements, which the system had been initially built with, making eventual consistency guarantees no longer appropriate for the business requirements. Given that systems tend to be built with an integrated consistency model, programmed within its core, making changes is much harder. Therefore, developers are left with two solutions: either a new system is developed or a deep restructuring is made to the current system code in order to fulfil the new business requirements.

Our proposal to solve this problem is a framework that abstracts the implementation of a system's consistency underlying model, making consistency model adaptations an easier task. To achieve this, we propose extracting the consistency implementation to a modular layer under the system, which is modelled so as to allow a shift of consistency model when needed. We believe that this proposal can also be useful for researchers, as it allows experimenting with variations in the consistency model in a simple way.

1.1 Contributions

In this document, we study the details of existing consistency protocols and seek common components between them. By splitting the functionality into different modules, we can create a framework architecture that is capable of being used to implement most of the existing consistency protocols.

This framework splits up the consistency implementation from the system itself, which allows reducing the effort necessary to add consistency to a system by abstracting the development process with the implementation of the well-defined modules as well as allowing the system's consistency to be changed at build time.

To the best of our knowledge, there is no similar solution to our proposal.

Briefly, this document makes the following contributions:

- A modular abstraction of the consistency model capable of being framed in most existing consistency protocol implementations;
- A framework that:
  - is capable of being used to implement the most common consistency protocols;
  - allows the developer to change the consistency of the system in build time.

1.2 Thesis Outline

In the following pages, we present the theoretical context which will help better understand the following chapters, and we describe the state-of-the-art by analyzing and detailing the consistency protocol of some representative existing systems in Chapter 2. In Chapter 3, we present an overview of our framework design and the details of the respective modules. We show the experience and details of the framework implementation in Chapter 4. In Chapter 5, we describe the evaluation of our framework and an analysis of the results. Last but not least, we present our conclusions and propose future work directions in Chapter 6.

Chapter 2

Related Work

In this chapter, we start by introducing a context of our thesis detailing replication techniques (2.1), next we describe a fundamental theorem (2.2), following by well-known properties (2.3), and consistency models (2.4) that are relevant to the understanding of our document. We ended up this chapter with a dissection of some the existing systems (2.5) to find out how each system achieves the consistency guarantees. In this analysis, we are interested in identifying the similarities between systems, especially in how they guarantee consistency in the system.

For this document, we only consider non-transactional operations, leaving transactional models out of the scope.

The change from vertical scaling to horizontal scaling due to the growth of the Internet and the complexity of service architectures created new challenges that needed to be faced. Keeping the system available while machines or network failures occur, or even a single machine being overloaded, unable to deal with a stream of incoming requests to this machine was one of these challenges. Horizontal scaling adds machines to a system that can be distributed worldwide, bringing more resource for more processing power, but also to add more reliability to the system. However, for this, all machines must have the same data as the others had.

2.1 Replication

All distributed system should be prepared to scale with the growth of the workload for which the system is exposed. However, to provide a system that is reliable, correct and with fault-tolerance guarantees, replication is mandatory. A good example of a replication need is a catastrophe scenario, such as a fire or an earthquake, that results in full destruction of

an entire data centre. Every piece of data should be saved on other data centres in order to be recovered. In some cases, replication adds the benefit of reducing the data access latency when the data are replicated to data centres near the client [9–11].

Replication could be performed in different ways: active replication (2.1.1) or passive replication (2.1.2).

2.1.1 Active Replication

A concept introduced by Lamport, under the name of State Machine Replication [12–14], in which each request is processed by all replicas with the same order in all of them. In order to guarantee that every replica receives the same sequence of operations, it's necessary to use an atomic broadcast protocol [15]. The atomic broadcast protocol guarantees that all the servers receive a message likewise they receive the messages in the same order. One drawback of this design is the high resources usage, such as CPU and network, that is required for each request.

2.1.2 Passive Replication

Contrary to the previous point, in passive replication, each request is processed by a single machine (primary) and then replicated to other machines. This approach is typically referred to as Primary-Backup [14, 16, 17]. This approach follows the master/slave model in which the primary machine with the master role is responsible to coordinate the replication to the slave machines. One request is replied to the client when a master machine has completed the replication to the slave machines. This design allows to read operations to be performed on any machine integrating the system.

2.1.3 Lazy Replication

Close to Passive replication, the differences between both approaches, allows the system to provide better performance sacrificing the consistency among replicas. In Lazy replication [18, 19], instead of waiting for replication to be completed before replying to the client, it is applied locally at a master machine and immediately replied to the client. Then the master machine initiates, in the background, a replication process by gossip protocol [20].

Replication is also concerned with data placement, that is, deciding where a data object is replicated to. There are two majors approaches:

2.1.4 Full Replication

This model allows the system to provide a higher availability since in full replication all data is replicated to all nodes. It means that all nodes on a system have a complete copy of the entire database [21].

2.1.5 Partial Replication

In contrast with the previous technique, partial replication [22] allows the system to have different data subsets replicated to different nodes, which are usually geo-distributed. The number of nodes with a copy of the data subset can vary with the importance of the data itself.

Ideally distributed systems should have strong consistency, high availability and partition-tolerance. However, the CAP theorem claims that it isn't possible to achieve at the same time.

2.2 CAP Theorem

Introduced in 2000 by Eric Brewer [23], and later proved by Seth Gilbert and Nancy Lynch [24] in 2002, the CAP theorem has become a reference in the distributed systems area. CAP is an acronym where we have:

- C for **Consistency** - if a client makes a write to a node, a following read will return this value or a more updated one. That is, the system ensures that the client never sees old data.
- A for **Availability** - every request receives a response without exposing errors to the client. This means that no operation can block indefinitely or return an unavailable state [25]. This property doesn't guarantee that the node, that is processing the request, remains in the most updated state.
- P for **Partition-tolerance** - a partition is a communication break within a distributed system. Partition-tolerance means that a system must continue to operate, regardless of if messages are dropped or delayed between nodes in a system.

This theorem states that from these three properties, only two can be achieved at the same time in a distributed system. Given that, there are three possibilities to the systems:

1. **Consistency and Availability (CA)** - systems that implement this approach agree that when a partition occurs, the system may be unavailable until the failure is solved. Usually relational databases systems use this approach, e.g., SQL Server [26], MySQL [27] or PostgreSQL [28].
2. **Consistency and Partition-tolerance (CP)** - this approach diverges from the above, in that, if some member of a system fails, a request may be rejected by the system. Usually, this happens because the system could reach a consensus. There are consensus protocols that aim to mitigate this problem, e.g., Paxos [29]. However, we will not describe these protocols in this document.

3. **Availability and Partition-tolerance (AP)** - systems with this approach neglect consistency, in order to achieve high availability and partition-tolerance. The decision behind using this approach is based on system performance, even agreeing that some inconsistencies may be exposed. Cassandra [30] or Voldemort [31] are some examples of systems based on these two properties.

2.3 ALPS

Many modern systems choose to provide availability and partition-tolerance (2.2) at the cost of consistency. Previously, ALP properties have been claimed to offer an "always-on" user experience system [32]. Adding scalability into account, it takes us to the ALPS properties, which are referred to as the desired properties for a geo-replicated service [25].

ALPS is an acronym for Availability, Low-latency, Partition-tolerance and Scalability. Since Availability and Partition-tolerance are already defined above (2.2), we will only describe the remaining ones.

- **Low-Latency** - all operations are completed quickly. According with Dynamo [32], a worst acceptable performance scenario is of 10s or 100s of milliseconds [25].

- **Scalability** - adding resources to a system increases its capacity in a behaviour approaching proportionality.

2.4 Consistency

Replication allows systems to no longer be fully centralized on a single machine and, in some cases, it also allows any replica to process the clients' requests, increasing the availability and throughput of a system. However, the advantages of replication have a price: the lack of consistency between replicas.

A consistency model determines rules for visibility and apparent order of updates to the system's objects [7]. Stronger consistency prevents exposing unexpected behaviour visible to users and reduces programming complexity [33] but at the same time, it has a huge impact on the overall performance. Acquiring locks, waiting for replication, or network delays are some of the problems that the system has to deal with that result in degrading system performance. This all culminates in a negative impact on the user's experience with the system.

Many production systems tend to choose weaker forms of consistency [11, 25, 32], in order to provide low-latency and high throughput. Shifting to the opposite side of the spectrum of consistency, the benefits described above disappear [33]. These weaker forms of consistency have two primary drawbacks: (i) The systems allow executions that expose anomalies to the user [11, 32, 33]. Example: someone in a distributed file sharing revokes the permissions of other user and upload new data, the person who had the permissions

revoked might be capable of to watch this new upload [11]; (ii) weakest models exposes problems that programmers must be aware and handle the complex cases, for example, the programmer must deal with a case where an album with references to photos that do not exist yet [33].

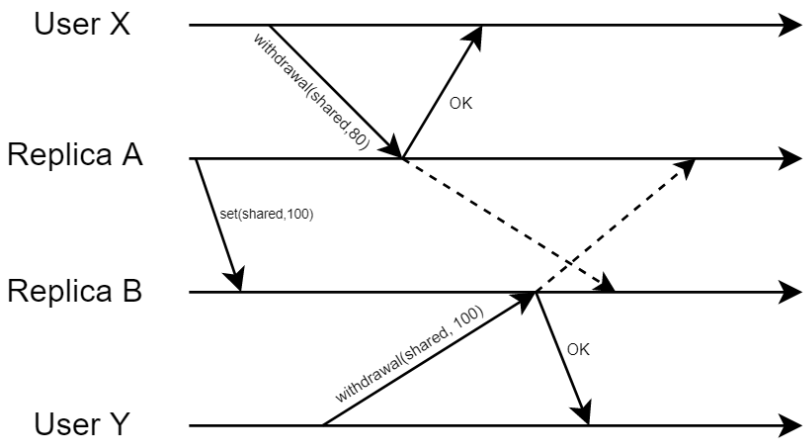

Figure 2.1: Example of an inconsistent system

Looking at the example of a distributed banking system, the consistency between replicas becomes critical. Say that a given user X has a bank account shared with user Y with a balance of 100 euros. X carries out a cash withdrawal operation 80 euros on an ATM. In another part of the world, Y withdraws 100 euros from the shared account. If the system is not consistently replicated, users X and Y are able to perform both operations and get more money than they had in their shared account (Fig. 2.1).

Dealing with replication operations requires providing certain guarantees for these same operations [33] in order to accomplish the system requirements. According to CAP theorem (2.2), if we have strong consistency, we are losing desired properties to our system. On the other side, if we have a weak consistency, it can cause the system to expose inconsistencies that do not fit the system requirements. To find a balance between the properties that can meet the systems requirements, several consistency models with different levels of consistency guarantees have emerged.

Before we describe the different consistency models, let's first split consistency into two ways of looking at it:

- **Server-side consistency** - how updates are propagated in the system and what guarantees systems can give about updates [34]. The replication technique used or which quorum is necessary to form to propagate and give a certain guarantee are some of the focuses here.

- **Client-side consistency** - how and when the clients observe updates made to a data object [34]. The strongest consistency ensures that two clients connected to different servers of the system, see the same latest update of a given object. However, using session guarantees (2.4.5), this cannot be ensured.

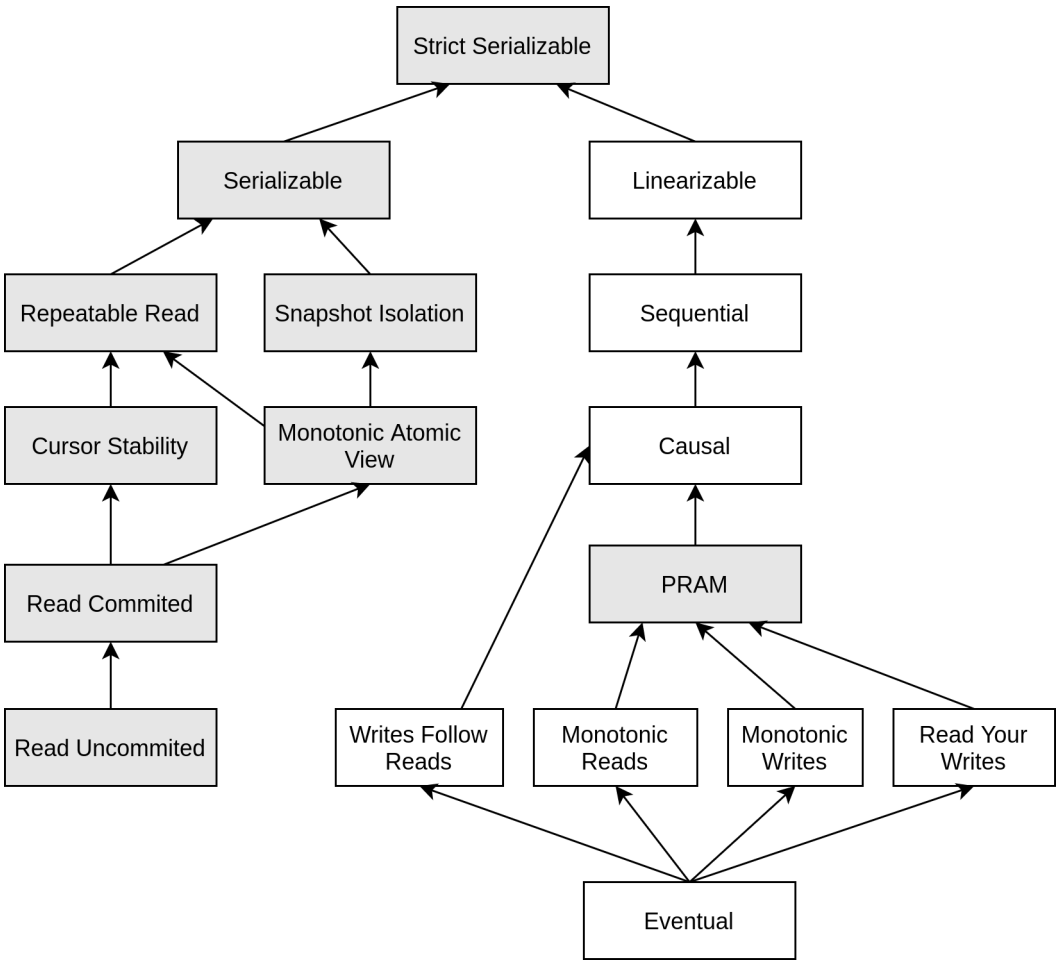

Figure 2.2: Consistency models tree, adapted from [35]

Looking at figure 2.2, it shows a tree that displays the consistency models in a top-down approach, which means that at the top of the tree is placed the strongest consistency model and at the bottom the weakest. Next, we will describe the consistency models following the top-down tree approach.

On the root of the tree, there is a Strict Serializable [36] model. It is out of our analysis for two reasons: it is a transactional model and, as we are not considering transactions, linearizability is just strict serializability for single object operations [37].

2.4.1 Linearizability

Linearizability is the strongest consistency level for non-transactional systems described by [33, 36], also known as atomic consistency, guarantees a total order with the real-time ordering of operations. This model guarantees that once a write operation is complete, a later read operation will return the value of that write or the latest value wrote. However, there is a detail: it supports concurrent operations. Given two concurrent operations in which A is a write operation and B is a read operation, there is no guarantee on what value B operation returns. B may return the value before A or the resulting value of A because

what this level guarantee is if an operation is complete, the following operation returns the latest value.

2.4.2 Sequential Consistency

This model guarantee total order to the system without real-time constraints. The result of any execution will be the same if all operation were executed in some sequential order in replicas and these operations respect the order specified by its program [38]. This model does not guarantee consistency between replicas. In other words, it is possible that different replicas be ahead or behind other replicas, but when a replica returns a state of an object, it is not possible for the same replica to give back a previous state that it already returned.

2.4.3 Per-Record Sequential Consistency

A weaker model following sequential consistency, also known as per-record timeline consistency. Instead of providing sequential consistency for the entire system, this model guarantees a per-object sequential consistency. The updates to an object have a single ordering processed by a single replica, according to a timeline, with that replica being responsible for propagating it to the other replicas. This means that all replicas of a given record apply all updates to the record in the same order [11]. That guarantees that a replica only moves forward on object versions and consequently a read operation is always consistent with the order.

2.4.4 Causal Consistency

This consistency model ensures that, if an operation B requires operation A to be correct, the system only applies operation B after applying operation A. Nevertheless, if an operation is not causally related to a previous operation then, these operations are concurrent and not ordered by causal consistency, then A can be immediately applied [8, 25].

This guarantees that if an operation is dependent on a subsequent operation, the system cannot return the result of the second operation without the first being available [8, 12].

The causal dependency is captured using the notion of potential causality, the happens-before ($\rightarrow$ relation) formally defined by Lamport [12] as:

1. Given two operations A and B that execute on the same process, if A was executed before B, there is a causal order between A and B, then A$\rightarrow$B.

2. Given two operations, where A is a write, B is a read operation and both operations can be executed at different processes, if B returns the value written by A, then A$\rightarrow$B.

3. The relations are transitive. Given three operations A, B and C, if A$\rightarrow$B and B$\rightarrow$C, then A$\rightarrow$C.

2.4.4.1 Causal+ Consistency

Initially introduced by Bayou [39] and PRACTI [40] and more recently revisited by COPS [25], Causal+ is essentially a Causal consistency with a convergent conflict handling. In Causal consistency, when two concurrent operations (non-causally related) are writes to the same object, they might be in conflict. Causal+ adds to the causal consistency a new guarantee: replicas never permanently diverge and conflicting updates to the same key are dealt with identically at all sites [25]. These properties are achieved by adding convergent conflict handling, using a handler function that makes all replicas deal with conflicting operations in a deterministic way. This results in clients only seeing progressively new versions of the objects.

2.4.5 Session Guarantees

A session, according to Terry's paper [41], is an abstraction of a sequence of read and write operations performed during the execution of an application. In a distributed system, for a given client, this same sequence of operations can be split and sent to different replicas, hence the result of this operations may not be consistent. The following consistency models aim to add additional guarantees to a given session. Contrary to the models already described, the session guarantees are guarantees only for clients.

2.4.5.1 Writes Follow Reads Consistency

Sometimes called session causality [42], in this model it is ensured that, if a write operation W follows a read operation R, then the write operation occurs on the value returned by R or on a more recent one [43].

2.4.5.2 Read-Your-Writes Consistency

This model ensures that, if a client performs a write, then the result of this operation must be always available to subsequent read operations. This model is only available to operations from the same client. If a client A performs a write, there is no guarantee that another client B performing a read operation receives the result of the last write from client A.

2.4.5.3 Monotonic Reads Consistency

This model ensures that, if a client performs two reads A and B, where B is done after A, the return of read B cannot be prior to the result already returned by read A. This model is also only available to operations from the same client. This ensures that given three operations: W is a write operation, R1 is a read operation and R2 is a read operation after R1, executed by the same client, if R1 returned the value of W then R2 needs to return also W [42].

2.4.5.4 Monotonic Writes Consistency

A replica should apply all writes by the order they were made. This means that a write, only can be applied if all the previous writes were already applied. This model is only available to operations from the same process and not from different processes ensuring that if a process make two writes, B after A, then all processes will see A before B.

2.4.6 Eventual Consistency

Eventual consistency is the weakest model represented on the tree of figure 2.1. In contrast with the previous point, when an update arrives at a replica, it overwrites the object without checking any dependencies. The guarantee that eventual consistency specifies is: If a replicated object stops being updated, eventually all replicas would have the most recent version of the object [8, 34]. Therefore, this consistency level does not guarantee that a read done after a write operation returns the most recent value.

Besides the models discussed, hybrid models have become common. Given that, sometimes some operations require a stronger consistency and others require performance, hybrid models use more than one model described above to form a new mixed model. An example of a hybrid model is RedBlue Consistency [9], where they use eventual consistency for one type of operations and linearizability for another type of operations.

2.5 Existing Implementations

In order to find out common components between consistency protocols that will allow us to build a generic consistency framework, in this section, we will analyze the implementation of several existing systems focusing on how the system is composed to guarantee the level of consistency they promise.

For this analysis, we chose different systems that lie on disparate regions of the consistency spectrum.

2.5.1 Dynamo: Amazon's Highly Available Key-value Store

A particular problem for the Amazon company was the need for an "always-on" system that could keep working even though an entire data centre is destroyed [32]. To accomplish this, the authors present Dynamo, a high availability key-value storage system that sacrifices consistency in order to achieve availability and partition-tolerance (2.2). Dynamo is decentralized, scalable, symmetric (all nodes have the same responsibilities) and supports heterogeneity (which means that Dynamo distributes the work amongst different nodes with different capabilities). Dynamo provides eventual consistency with partitioned and replicated data using consistent hashing [44]. Data consistency is achieved with object versioning. Consistency between replicas is maintained by a quorum technique and a

decentralized replica synchronization protocol. For failure detection and membership a gossip-based protocol is used.

To allow the system to be scaled and load-balanced with potential arrangement changes, they dynamically partition data over the nodes and make use of consistent hashing [44] to select a node from the "ring" arrangement that receives each data item.

Looking at replication, Dynamo replicates data to N nodes (with N being a configurable value). The node that receives the data item is responsible for replicating the data to N-1 neighbour nodes in the ring asynchronously.

As a result, put operations can return before they are applied at all replicas and therefore get operations may return an outdated version of the objects if the return is from slow nodes. In order to solve inconsistencies, Dynamo uses vector clocks to capture the data versioning while treating every modification of the data as a new and immutable version. In case of conflicting versions of an object, versions can be reconciliated. Dynamo exposes two operations: $put(key, context, object)$ and $get(key)$, where the context parameters encode the metadata about the object and the version of the object. When a client wants to update an object, it specifies the version of the object that he wants to upgrade. When a client wants to get an object, all different versions are returned to him and it is his responsibility to deal with.

2.5.2 PNUTS: Yahoo!'s Hosted Data Serving Platform

PNUTS is a massively parallel and geographically distributed system for Yahoo's web applications, presented to accomplish Yahoo's requirements which are Availability, Low-Latency, Partition-tolerance, and Scalability properties (2.3). The requirements are the same as Dynamo's. However, for PNUTS, Dynamo's consistency is considered too relaxed. Take a look at the following example: if, in a distributed file sharing, user X revokes user Y permissions and uploads new data, the person that had the permissions revoked (Y) shouldn't see this new upload. However, this is possible under Dynamo's consistency model. The authors agree that it is often acceptable to read stale data, but, on some occasions, stronger guarantees are required. PNUTS has per-record sequential consistency but does not provide consistency between different records. So, in PNUTS, this variation is called a per-record timeline consistency.

PNUTS relies on Yahoo! Message Broker (YMB), a publish/subscribe system, to take care of all asynchronous replication and provide failure recovery to the masters. It performs a full replication to each node of the system. Another difference compared to Dynamo is that PNUTS provides a middle approach between decentralized and centralized. It means instead of having one global master or one full decentralized system, PNUTS designates a master per record that is responsible to serialize that record. This master is also responsible for executing all updates on a given record.

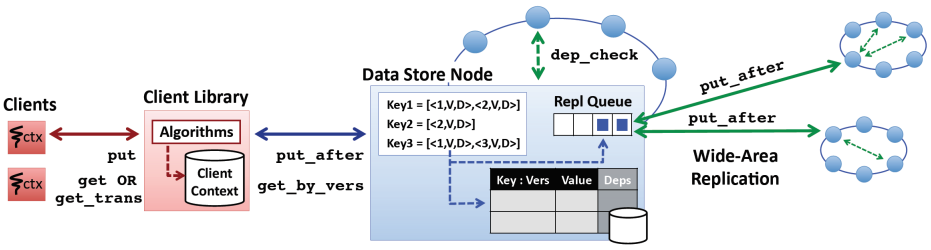

Figure 2.3: COPS architecture (from [25])

2.5.3 Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS

The eventual consistency that Dynamo [32] or Cassandra [30] expose is too weak to guarantee that all replicas are consistent. Hence, the authors of COPS proposed a causal consistency model with convergent conflict handling called causal+ that achieves the ALPS properties (2.3). The convergent conflict handling solves a problem of causal consistency for concurrent events. Applying last-write-wins [30] rule ensures that all conflict problems are handled consistently at all replicas. However, other conflict handling rules can be implemented, for example, first-write-wins.

It is not the first appearance of causal+. Systems as Bayou [39] or PRACTI [40] achieved causal+ using log-based replay from a centralized point. However, these systems were not scalable since they required that all data (or at least data that might be accessed together) fit in a single machine. For that purpose, the authors introduce COPS: a key-value distributed storage system (Figure 2.3), in which data can be spread across many machines and multiples data centres. Every key has the following format: $< version, value, dependencies >$. The reason for including dependencies in the API is to guarantee causal order for each key's version when a message is applied. COPS is designed to work across spread data centres where each has a local COPS cluster. This cluster uses consistent hashing [44] to partition the keys across cluster nodes, assigning different nodes to different keys. On the client-side, COPS provides a client application that uses the COPS client library (with put and get operations) to make calls directly into the COPS data store. The `get` operation is non-blocking, and it is performed locally by the nearest data centre since all data are replicated. When a client calls the put operation, the primary node of a cluster assigns a version number to each update using a Lamport timestamp [12] and returns it to the client. Then, the primary node replicates the update asynchronously to the other data centres. The message is applied to the replicated data centre when the dependencies are satisfied. It means that operation dependencies required are already applied to the system. The operation result returns to the client after being executed in the local cluster, and the following operations between cluster occur asynchronously in the background.

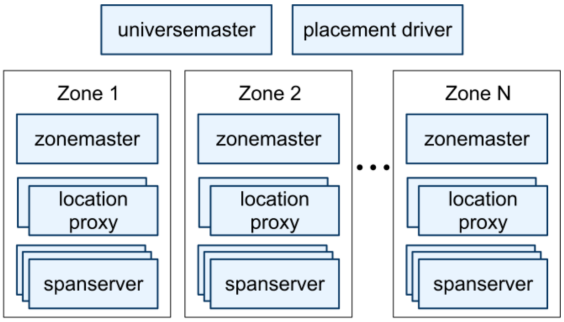

Figure 2.4: Spanner Architecture (from [45])

2.5.4 Spanner: Google's Globally-Distributed Database

Spanner [45] is a scalable distributed database introduced by Google that provides externally consistent (similar to Linearizability [46]) reads and writes, and replicated sharded data across data centres spread all over the world providing globally-consistent reads across the database at a timestamp.

To guarantee these properties, Spanner makes use of the TrueTime API. This API not only provides a global clock but also directly exposes clock uncertainty with the particular concern that it is able to give a real time uncertainty between 1ms and 7ms. Spanner was built around TrueTime timestamps. If the bounds of uncertainty are high, Spanner slows down the execution to wait out that uncertainty.

A Spanner deployment is called a *universe*. Inside the *universe*, there is a set of *zones*, which are locations (could be data centres) where data can be replicated. Each zone has one *zonemaster* and between one hundred and several thousand *spanservers* (Figure 2.4) where the *zonemaster* assigns data to *spanservers* and the *spanserver* is responsible for provide data to the clients. Each *spanserver* is responsible for between 100 and 1000 instances of a *tablet*. A *tablet* is a bag of mappings between a (key, timestamp) pair and a value.

Let's now focus on analyzing the consistency model. The applications can choose to which zone they want to replicate. To enable replication, each *spanserver* implements a single Paxos [29] state machine on top of each *tablet* (Figure 2.5) that is used to implement a consistent replicated bag of mappings. Each Paxos state machine stores metadata and a log in its corresponding *tablet*. At every replica, which is a leader of a zone, each *spanserver* implements a lock table, and a transaction manager to support concurrency control and distributed transactions.

Given that, in this thesis, we are only considering non-transactional models, for our analysis, we are going to consider a transaction over a single object. Spanner implements the following transactions: Read-Write Transactions and Read-Only Transactions.

Read-Write Transactions makes use of a two-phase commit protocol [47]. First, the client makes a read request to each leader of the group that acquires the locks and read the latest data. Then, the coordinator's leader acquires write locks and receives all the timestamps from the other leaders, chooses a timestamp s that is greater than all of them,

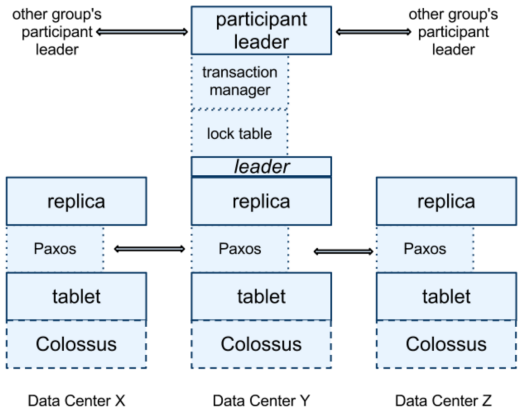

Figure 2.5: Spannerserver Software stack (from [45])

sends s to all other leaders and releases the locks. To guarantee the external consistency, the coordinator's leader ensures that clients cannot see any data committed with timestamp smaller than timestamp s.

Read-Only Transactions are performed in two ways: if the read is at a single group, then it assigns a read timestamp equal to the latest committed write in the group. If the read is along with multiple groups, a timestamp from TrueTime is assigned and the operation has to wait until this timestamp is exceeded.

2.5.5 Bolt-On Causal Consistency

There are many well-tested projects which the developers have spent a lot of time refining their solutions. Taking advantage of that fact, the authors implemented a layer to guarantee causal consistency (bolt-on architecture, fig. 2.6) to the underlying eventually consitent data store (Cassandra [30]). One of the reasons for this design choice is to leave the responsibilities of liveness, replication handling, durability and convergence for many well-tested projects, and leave to the implemented layer the responsibility of providing causal safety guarantees.

There is a problem of trying to implement causal consistency on top of an eventually consistent data store. While eventual consistency usually overwrites previous values with more recent writes without checking for dependencies between operations, causal consistency needs to define the dependencies of each operation which is only achieved by storing each version of an object. It will allow building the previous dependencies for each new operation in a system.

The solution presented is the notion of *causal cuts*, that defines the dependencies of each write with a cut in the operations history (Figure 2.7). A causal cut object is defined according to the following the rules: (i) to be in the cut; (ii) to happen-before a write to the same object that is already in the cut; (iii) to be concurrent with a write to the same object that is already in the cut.

When a client calls a write operation, the layer updates the local store and sends it to the data store with the dependencies list that must be ensured before exposing the write

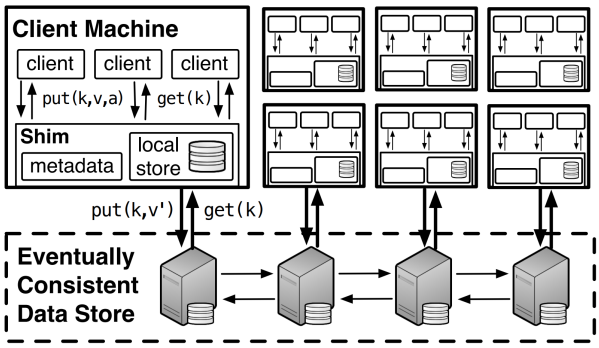

Figure 2.6: Bolt-on architecture: a causally consistent shim layer mediates access to an underlying eventually consistent data store [48]

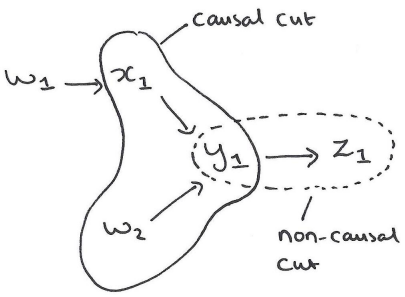

Figure 2.7: Causal Cuts [48]

to the clients. The clients can only read from the local store. So, if a read is already in its local store, the value can be immediately returned. If the value is not up-to-date, it can be updated asynchronously.

2.5.6 Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary

In this paper, a new consistency model called RedBlue is proposed as well as a system, called Gemini, that implements that model [9]. The main concept of RedBlue is: no matter what the operations are, all replicas must converge on the same final state and, at the same time, ensure that application invariants are never violated. For this, the operations are classified into two types: red and blue. Red operations use a strong consistency, being serialized and immediately applied to all replicas. Blue operations are operations which can be executed after or before any other operation because they don't potentiality violate any invariants. Blue operations execute on the local replica with eventual consistency.

Ideally, applying RedBlue consistency to an application, all operations should be blue to obtain the best performance. However, when operations are not commutative, this could lead to invariant violations or state divergences [49]. To bypass this, the authors decompose each operation in two phases. The first one consists in using a generator operation that simulates the changes that operation would cause, producing what they define as *shadow operation*, locally. On a second phase, the *shadow operation* is executed on every replica. These are the only operations that use the classification blue or red.

Gemini trusts each local site to replicate the operations to all remote sites. On that system, the generator operation assigns to the *shadow operation* an independent timestamp for each operation colour. These timestamps are standard logical clocks [12]. To ensure that different sites do not choose the same red sequence number, the coordinator holds a unique token that is used to approve red operations. On blue operation, the same sequence number might be assigned to multiple operations, if they are executed at different sites.

2.6 Discussion of Existing Implementations

Despite offering different levels of consistency due to varied operational goals, the analysis of the systems above shows that there are several points in common between them.

First, all the analyzed systems timestamp messages even if in different ways. Looking at Dynamo [32], they use vector clocks to capture the data versioning treating every new operation, as a new and immutable version. RedBlue [9] uses standard logical clocks to timestamp both types of messages. COPS [25] and Bolt-On [8], capture the version among with dependencies of each object and Spanner [45] timestamps the objects with real-time using TrueTime API.

Systems that timestamp messages at a single node, such as PNUTS [11] which defines a master per record, do not need to deal with inconsistencies given that the same object is only timestamped on a master node. Nevertheless, COPS [25] and Dynamo [32] follow a decentralized design that may need to solve inconsistencies, in which the first applies by default the last-write-wins rule, while Dynamo solves inconsistencies by reconciliation.

Second, objects should be replicated. All systems have to be aware of the nodes that form a system and to where they have to replicate to. On Spanner [45], the applications can choose where to replicate. Gemini [9] replicates to all nodes. Dynamo [32] replicates to N-1 nodes on a ring format, where N is a configurable value. As we can see, all systems, in one way or another, replicate to a set of nodes. Even if it is not well specified where to replicate, such as in the PNUTS [11] or Bolt-on [8] systems, they delegate this task to an external system. Thus, it is quite obvious that this task needs to be performed when a system is starting a replication process.

There are other points that we identified as being part of stronger consistency systems, but which are not part of weaker consistency systems. For example, an eventually consistent system does not provide replication guarantees, while a strongly consistent system does.

Another point, which is part of some of the above systems is that they might need to form a quorum or define a semantic (e.g. at least one) to consider messages as replicated. As an example, Red operations of RedBlue [9] that provide strong consistency have to wait for all nodes to apply the messages before continuing.

The last point that we identified as common to all systems, except for eventual consistency systems, are delivery conditions. We consider delivery conditions as a set of rules that should be ensured to be true in the system before the operation be performed. Looking at COPS [25] and Bolt-on [8], they provide a causal consistency that exposes dependencies to the objects. These dependencies need to be visible at a node before applying an update to an object. On Spanner [45] and PNUTS [11], it is necessary to check and wait until the message that preceded a new one already exists in the system.

Briefly, there are some points that we identify as being common to consistency protocols implementation which are: it needs to know how and where to replicate to, when to consider a message delivered at a node, how the quorum is formed, how to timestamp a message and what to do in case of conflicting messages.

Chapter 3

Architecture

In the previous chapter, we identified and discussed the common components among several consistency protocols. In this chapter, first of all, we present our proposal that abstracts the implementation of a system's consistency, the underlying component architecture, which will allow us to make variations to the consistency system with the minimum necessary effort. In a second step, we are going to demonstrate the flow of a message in the system and its interaction with the modules, in order to guarantee the desired consistency.

We propose a framework, which exposes two different APIs: an external API that is exposed to the replicated system to communicate with the framework, and an internal API which is used to communicate between replicas using our framework. This framework accomplishes the following requirements:

- Modularity - the solution is divided into modules to allow possible future extensions to the framework, and to allow an easy swap of a module for another one of different consistency.

- Ease of use for developers;

- Generality - the solution has to be the more general possible in order to be used to implement as many consistency systems as possible.

From the analysis in section 2.6, we decided to split our framework into seven modules that will be individually described in detail below: Group Membership (3.1), Ordering (3.2), Replication (3.3), Delivery Condition (3.4), Quorum (3.5), Communication (3.6), and Framework API (3.7).

3.1 Group Membership

This module is responsible for managing all the information about which nodes participate in the system. The members need to specify the roles that they perform within the system.

We define two membership types: Timestamper and Forwarder. A Timestamper acts in a system as a member that is capable of marking new messages with a timestamp. A Forwarder role acts as a slave, which means that if this type of member receives a new message, he has to forward the message to a member with a Timestamper role. In a fully decentralized system, all members could behave as timestampers [32]. For systems that use leader election, there is also the possibility of implementing a leader election on top of this module or even to coordinate the operation involving multiples nodes [45].

This module exposes four methods:

- *getMySelf()* - returns information about the own node caller of the method. Information such as the own role, which can be either timestamper or forwarder, or the data centre ID or the partition ID to which the node belongs is some of the data that this method returns.
- *getReplicationTargets()* - returns a list of members to where a member must replicate a message to. The return of this method is member dependent because one member could replicate to all others or just to some (gossip schema).
- *getTimeStamper()* - returns a timestamper member of the system.
- *findPartition(key)* - returns the partition ID that a given key belongs to.

3.2 Ordering

This module has three different roles: first, it is responsible for holding the timestamping mechanism, which could be a logical clock, such as Lamport clock or a vector clock, or even a physical clock. The second role is timestamping messages, assigning an order to the messages that arrived at the framework, provided by the *timeStamping(content)* method. Lastly, the third role is to compare messages and to define an order for messages with the same timestamp, provided by *compareMessages(message1, message2)*. It is this module that provides conflict handling.

Every message that is processed by one member needs to be marked with metadata, in order to be distinguished from messages forwarded by other members of the system. This metadata is a key-value map that contains all the additional information of the message. However, there are some entries that are more common, such as version resultant of *timeStamping(content)* method execution, message origin source or even progress status of the message in our framework, like the number of replication successes. Optionally, some consistency protocols, such as RedBlue [9], add information about the type of the message (Red or Blue) to the metadata.

Causal consistency systems, for example, require that a dependencies list be generated. This feature is also provided by *timeStamping(content)* method at the time of execution, and saved into message metadata.

In addition to the previous methods, this module also provides:

- *updateClock()* - method responsible for incrementing the actual clock.
- *updateClock(newClock)* - take the newClock value, and replaces the actual clock value or uses it to update the existing one.

3.3 Replication

This module is the core of replication of our framework, and is responsible for coordinating message replication. It provides two methods:

- *replicate(content, metadata)*
- *apply(content, metadata)*

In order to be possible to replicate a message, in the *replicate(content, metadata)* method there are interactions with the Group Membership (3.1) and Quorum (3.5) modules. These interactions allow a member to know where to replicate, *getReplicationTargets()* (3.1), and when a message can be marked as replicated, *waitQuorum()* (3.5).

The *apply(content, metadata)* method is called when it is necessary to apply the message to the system member that received it. This method is supported by the Delivery Condition (3.4) module.

There are two considerations that we have taken into account. Some replication types using gossip mechanisms require that after receiving and applying a replication message, that message should be replicated to other members. This scenario has been considered and calling the *replicate(content, metadata)* method inside the *apply(content, metadata)* method it is possible. Lastly, we do not force an order onto the message pipeline. For example, a system could first apply the message, then answer to the client and only after this start the replication process. However, it is also possible that the framework has to wait for the apply and replication process, before it answers to the client. Briefly, applying a message to the system can be done when a message has already been replicated, or when it has not yet been replicated (and it may, or may not be in the future). This is a dependent system choice.

3.4 Delivery Condition

This module has defined the conditions that need to be satisfied to apply a message to the system and consequently consider the message as delivered. It may need to use the Ordering (3.2) module to compare messages and decide if the message may be applied or not. For

example, when a system is trying to apply a message B that is causally related to message A, it is necessary to check if the message B can be applied. In cases where all conditions, initially defined, are not satisfied, the system must wait until the defined conditions are all satisfied before applying and returning. This is provided by the *tryToApply(content, metadata)* method.

Depending on what system we are working with, it can have very different delivery conditions. It is possible that the consistency system only cares about temporal occurrences, in which case it only checks if the message that is trying to apply occurred after the existent one. However, causally consistent systems demand a way to deal with dependencies. Local dependencies to the system should be treated and managed into this module. Nevertheless, when not all dependencies are satisfied, the system can choose to request it to other members. To accommodate this requirement, it also provides two additional methods:

- *addToRemoteWaitingDep(dependencyRequest)* - this method adds a remote request of dependency to a queue to be answered when satisfied by a local system.

- *removeRemoteWaitingDep(message)* - This method removes a dependency request from the waiting dependencies queue, that was been answered by another member. It should be called when a response to a dependency request arrives at the framework.

3.5 Quorum

In order to consider a message as replicated, some consistency models demand a quorum. In this module, it is possible to implement quorum algorithms and/or define the semantic required (e.g. at least one read or write, or even number of zones required). Only one method is provided: *waitQuorum()* that is responsible for guaranteeing that the other members have already returned a delivery status message and a quorum has been formed.

3.6 Communication

We decide to split the communication module into two parts: internal and external communication. By internal, we mean that communications take place within the member, such as a call to write or read on a local database, whereas by external, we mean that communications occur from the member to the outside. For example, replicating to other members or answering to a client.

It is the programmer's responsability to implement this module and it is system dependent, so all the following methods need to be implemented by the programmer. The reason behind this module creation resides in the necessity to convert data type between the system below and the framework itself, and between the framework and the real communication protocol chosen by the system and/or programmer.

3.6.1 Internal communication API

- *get(node, content, metadata, callback)*
- *put(node, content, metadata, callback)*
- *delete(node, content, metadata, callback)*

These three methods above should implement a call to a database.

- *getActualVersion(content, metadata, callback)*

Returns the current version for a given key defined in *Content*. Note that although this method is located as internal, some consistency protocols may request other members in order to obtain the most current version.

3.6.2 External communication API

- *sendGetResponse(node, content, metadata, callback)*
- *sendPutResponse(node, content, metadata, callback)*
- *sendDeleteResponse(node, content, metadata, callback)*

These three methods above should implement the behavior in case of responding to a request of each type above. For example, answer to a Node get request.

- *replicate(node, content, metadata, callback)*

This method is responsible to send a replication message to other *Node* of the system.

- *sendDependenciesCheck(node, content, metadata, callback)*
- *sendDependenciesResponse(content, metadata, callback)*

The two methods above are specific to systems with dependencies mechanism, as causally consistent systems. These systems need to send requests to dependencies missing and answer for the requests from the other members of the system. These methods define how this occurs.

3.7 Framework API

Our framework exposes two APIs, a public API for applications and a private API for the communication between nodes. We start by describing the two methods that compose the public API:

- *newMessage(content)*
- *newMessage(content, metadata)*

These methods are used when a new message is arriving at the machine. Some consistency models as [9] require providing additional information about the message. For this, we decide to provide the possibility of optionally submitting messages with some metadata.

The next methods below belong to the private API, which we decide to expose two methods:

- *replicateMessage(content, metadata)*

When a member wants to replicate a message that is already marked with metadata. This means that the message was already processed by another member.

- *getReplicaState()*

This method allows one member to get the actual status from other members. It could be used by some systems to synchronize the members, typically in systems where servers need to exchange state information [50].

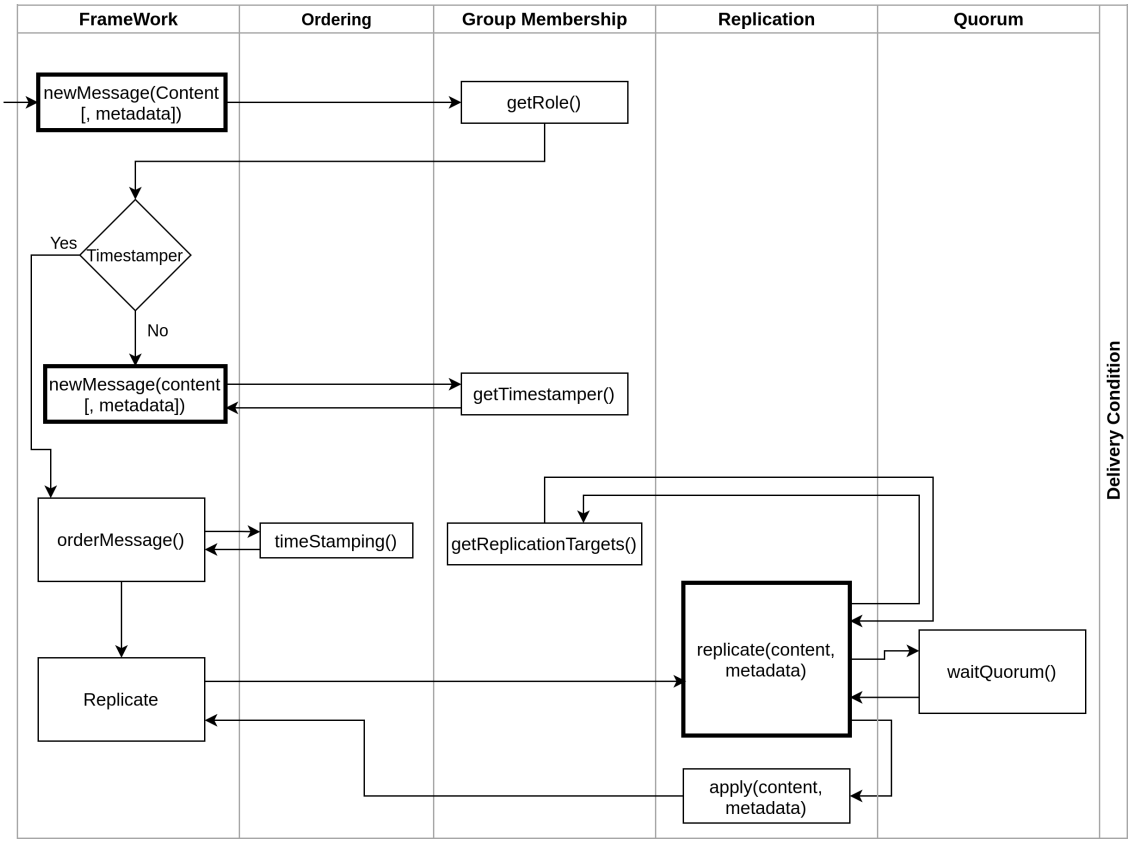

Figure 3.1: New message flow

3.8 Inter-module interactions

To better understand how this solution works, we are going to describe next the flow of a message within the system detailing how the various modules interact each other. Given that our solution is modular, which allows the developers to change the course of a message

and each consistency system has different choices, the following descriptions represent a possible execution. Note that we do not describe the interactions of the other modules with the communication module for simplification.

We separate the description in two different events: a new message (1) and a replicated message (2) in the system.

1. **New Message** Figure 3.1 shows the interaction between modules when a new message arrives at the system. A new message arrives at the system via a call to the *newMessage(content)*, a method exposed by the API (3.7). To decide what to do with the message, first, the respective replica which received the message invokes the *getRole()* method on Group Membership module (3.1) to check what is its own role on the system. Then, it checks the result of the last invoked method and decides what to do.

   In case the receiving node is a forwarder, it should forward it to another member with a Timestamper role. This is achieved by invoking the method *getTimestamper()* on the Group Membership module and forwarding the message invoking the *newMessage(content)* on the replica that the *getTimestamper()* method returns.

   If the node is itself a Timestamper, it proceeds to order the message in the system. The order of a message is provided by the method *timeStamping(content)* of the Ordering module (3.2). For example, if it is a causal consistency system, the method will check and return the dependencies of the message inside the metadata field. Subsequently, the message is marked with a timestamp, and it is ready to be replicated. So, the message is passed to the Replication module (3.3) by the *replicate(content,metadata)* method to initiate the process of replicating a message.

   The Replication module has to know where to replicate. Thus, it invokes the *getReplicationTargets()* method on Group Membership module that returns a list of members to where it must replicate and then, it calls *replicateMessage(content, metadata)* on each member of the list. Some consistency models require the system to wait for the replication response before considering the message delivered. The *waitQuorum()* method of the Quorum module (3.5) only returns when the replication conditions are satisfied. Finally, the message that was replicated needs to be applied to the local replica by the *apply(content, metadata)* method of Replication module. It may also needs to use the *tryToApply* method of the Delivery Condition module (3.4), which is not detailed in figure 1, but in figure 2.

   There are two scenarios for applying a message to the system: (1) we wait for the replies from all the targeted replicas before applying the message to the system. This is the case from the aforementioned strongly consistency scenario; but we can also, (2) apply the message right away and asynchronously replicate to targeted replicas, which it is the typical case of eventual consistency systems.

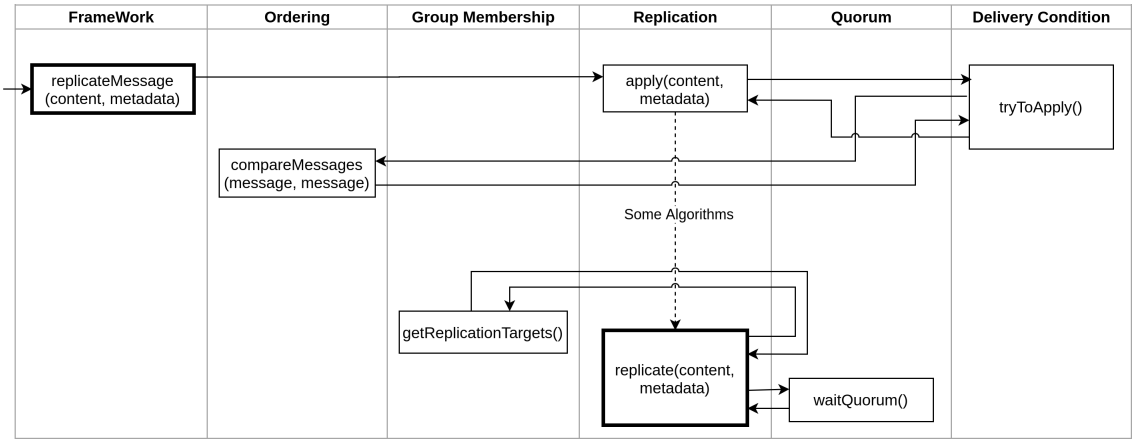

Figure 3.2: Replication Message Flow

2. **Replicated Message** A replicated message process is initiated by the Replicate module (3.3), which invokes the API *replicateMessage(content, metadata)* method on the respective replicas to which a node wants to replicate a message. Figure 3.2 shows the interaction between modules when a replicated message is received by one replica.

   Given that the system distinguishes between messages being replicated and new messages in the system, a replicated message does not need to be timestamped again. Thus, it invokes the *apply(content, metadata)* method in the Replication module to initiate the process of applying a message to the system. In order for this to happen, the *tryToApply()* method of the Delivery Condition module (3.4) is invoked, which will ensure that all conditions are gathered for the message to be applied. In some cases, the node receiving a replication call may need to invoke the *compareMessages(message, message)* method in the Ordering module (3.2) in order to solve conflict cases.

   Some consistency protocols that make use of a gossip propagation schema, after the message is applied, must replicate to other replicas. In this case, the *replicate(content, metadata)* method of the Replication module is invoked, which initiates a replication process described in the previous point, without the apply message part which already had been performed.

Chapter 4

Implementation

4.1 Methodology

To implement our architecture, we explored two possible approaches: (i) building a system from scratch with our framework built-in, and choosing a consistency model that this system will provide. (ii) modifying an already existing system by programming our framework to have the same consistency model that the system already implements.

Both approaches have advantages and disadvantages. Looking at (i), building a system from scratch will give us the flexibility to decide all components of the system. But, this will take time to build and debug until all performance, and consistency model requirements chosen are satisfied. In contrast on the (ii) approach, we start with an existing tested system where we have all the parameters defined, making us limited to the options that it offers.

Looking at it in a more particular way, and given the nature of our proposal, we decide to follow the (ii) approach. The first reason behind this decision is that by modifying an existing system to implement our framework, it leaves us the opportunity to see and learn how our proposal fits a real system, and not only build a new system around it that would necessarily fit the framework. The other reason is that with this approach, we are able to compare the real impact of our framework by comparing the original implementation of an existing system against our modified version, which implements our framework. These two reasons are not possible to achieve following the (i) approach.

4.2 Programming Language Choice

We decide to use Java as the main programming language for the implementation of our framework. As Java is one of the most used languages in the world [51], there is

a lot of systems and tools available that we could use. Nevertheless, as it is an object-oriented language, it seems to be a good choice for a better code organization and simple understanding of the produced code. Lastly, regarding memory management, we could have chosen a more efficient memory management language like C. However, this adds a lot of concerns that do not will add any significant aspect to our work.

4.3 Choice of replicated storage systems

Given the reasons discussed in the previous sections, we decided to choose two different systems with the following criteria: the code should be written in Java, not offer the same level of consistency, be open source and be available in some online repository.

In the next two subsections (4.3.1 and 4.3.2), we will describe each of the chosen systems and the configurations/variations considered.

4.3.1 DKVF

Distributed Key-Value Framework [52], is a framework that allows programmers to quickly create and evaluate distributed key-value stores. DKVF based systems offer the client and the server-side that extends the client and server-side DKVF, respectively. The code is written in Java, and it relies on Google Protocol Buffers [53] for marshalling/unmarshalling data for storage and transmission. Although DKVF can use any storage engine, it already comes with a driver for Berkeley-DB which can be configured to handle the data replication. We do not use this functionality, leaving the replication to our framework.

A DKVF client exposes two basic operations: put and get, to keep the interface simple. Yet, it is possible to extend the framework to use other methods.

From source code available on GitHub [54], where there are some systems implementations using DKVF available, we decide to use the COPS [25] implementation. Given that it offers a causal + consistency that needs to deal with dependencies, it will increase the complexity of some framework modules and give us better feedback of the framework fit into this system. In this COPS implementation, the client is only responsible for sending the messages to the server along with a list of dependencies for a given key, and the server will ensure all the consistency and replication process.

4.3.2 Project Voldemort

Voldemort [31] is a distributed key-value storage system, used in critical services at LinkedIn [55] based on Amazon Dynamo [32] architecture. In order to keep the high performance and availability, Voldemort only supports four queries to the data access: put, get, getAll and delete operations. Although Voldemort has different consistency guarantees out-of-the-box in the source code, we decided to choose the implementation with eventual consistency. This solution could lead to inconsistencies, but to mitigate this problem, Voldemort tolerates the possibility of inconsistencies, and resolve them at read time. The approach is called

read-repair, it consists of writing all inconsistent versions and at read-time detecting and solving the problems. To versionate the objects, vector clocks are used, which is a list of *server:version* pairs.

Contrary to the previous system, and although the available documentation says that it is possible to choose between who does the messages replication (being the client or the server), the Java implementation does not provide this option. It is confirmed by an issue closed [56] on GitHub. So, Voldemort offers a "smart" client that is aware of all the cluster, is responsible for replication and guarantees the system consistency.

4.3.3 Discussion

We choose these two systems because they have different approaches to the system design and because they offer different consistency guarantees. Whereas the COPS implementation using DVKF offers a causal+ consistency that needs to handle dependencies, Voldemort offers an eventual consistency with read-repair. Looking at our framework, it will produce, at least, significant different Delivery Condition modules, since the COPS implementation needs to ensure that all dependencies are satisfied in the system before considering a message delivered. Conversely, the Voldemort adaptation will produce a way more simple module.

When we first think about our architecture (3), we focus only on the possibility of server-side implementation. This scenario is what we found out on COPS implementation, where the entire consistency protocol is implemented on the server-side. However, Voldemort takes a different approach where the client is given an important role in the consistency guarantee. We concluded that implementing our framework on the server-side or the client-side will not produce any changes to it. Additionally, it is possible to implement it on both sides, if that makes sense to the system, where both sides have roles on consistency guarantees. However, it can lead to some of the modules being empty, but this possibility is allowed even in a single side implementation, e.g., a system that does not use quorum may have an empty quorum module.

4.4 Code Structure

Starting by the code structural division, we decided to divide it into 6 parts (Fig. 4.1): *main code*, *cluster*, *exception*, *types*, *versioning* and *utils*.

The *maincode* is where all modules code is placed along with the corresponding interfaces. There are also two important classes: Framework and Configurations.

Framework is the class that should be instantiated to use our framework. It is in that class that the incoming messages to the framework are processed and where the message flow is coordinated. The configurations class interacts with almost all classes of the system, and it has important variables like minimal number of writes to consider a replication

message delivered. In order to support asynchronous calls, we also provide a Callback interface that classes need to implement.

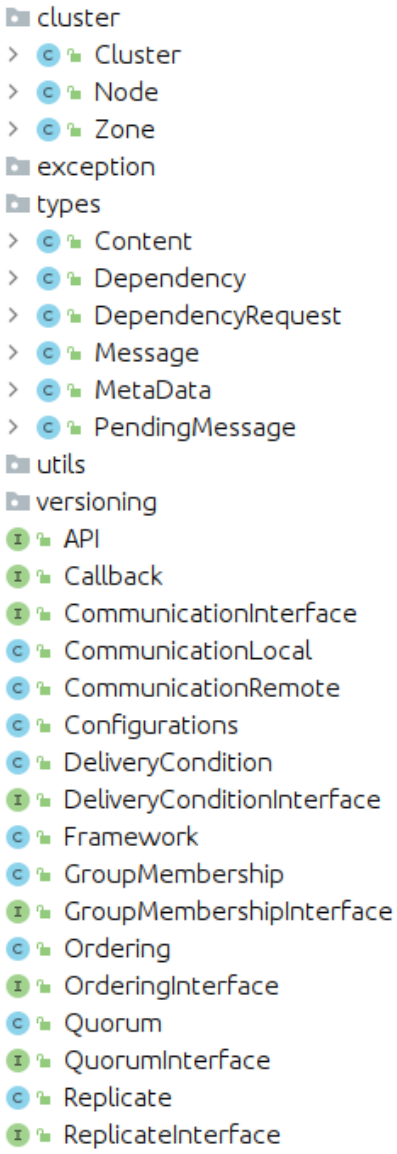

Figure 4.1: Framework code structure

Moving on to the *cluster* package, it encompasses all the notions of a cluster on our framework: a cluster, a zone and a node. A cluster consists of a number of zones and a number of nodes. A zone has its zone number along with a list of proximity zones, and a node has all information about a member of the system, such as id, IP address and zone number that the node belongs or partitions that the node has.

The *types* package has great importance. This package defines all basic types of our framework. A message has a type, which can be a put, get, dependency request, etc., and content and metadata. The remaining three types defined are relevant to deal with dependencies at the system. A dependency is a key-version pair, whereas a dependency request is used to store a request for a dependency from other nodes of the system. However, if a message has dependencies that are not satisfied, PendingMessage is used to help store messages that need to wait until all dependencies are satisfied.

The *versioning* package includes the version interface that classes that will be used to versionate objects must implement. Vector clock or Lamport clock are examples of clocks that could be placed in this package.

In the *exception* package all exceptions used within the framework are defined.

The last package, *utils*, has classes that help the programmers, e.g., serialization functions or time functions.

4.5 Main Class

As stated in the previous section (4.4), Framework is the main class of our solution. First, this class is responsible to instantiate all modules that will be used. Second, it is the entering point to the messages into our framework. Lastly, it is in this class that the main flow of a message is defined.

Listing 4.1 shows an implementation of a new put incoming message into the framework from COPS using DKVF (4.3.1).

```
public void newMessage(Message<K, V> incomingMessage) {
    long startTime = time.getNanoseconds();
    Content<K,V> content = incomingMessage.getContent();
    MetaData metadata = incomingMessage.getMetaData();
    switch (incomingMessage.getType()){
        case PUT:{
            if(isTimestamper()){
                orderMessage(content, metadata, startTime);
                replicate.apply(content, metadata);
                communicationRemote.sendPutResponse(content, metadata, null);
                replicate.replicate(content, metadata);
            }else{
                int timestamper = groupMembership.getTimestamper();
                ...
            }
            break;
        }
        ...
    }
}
```

Listing 4.1: New put message on framework

This code block is in accordance with Figure 3.1 that is detailed in Section 3.8. However, this system answers to the client before replicate. It could have implemented other sequences, e.g., replicate before apply, and subsequently answer to the client. Our solution makes changing this sequence as simple as changing the order of the code lines.

4.5.1 Switching modules and versioning

The Framework class instantiates the modules that will be used on the execution of our framework. The modules that will be instantiated are defined in the Configurations class. Each of these modules can be exchanged for the same type of module with different implementations. For example, if we have two systems implemented differently with our framework, one with causal consistency and another with eventual consistency, we can switch the Delivery Condition module of the causal system to the same module of the eventual system. In this case, a delivery condition module from a causal system should check and try to satisfy dependencies on the system. However, the change to a delivery condition from an eventual system removes all the dependencies checking from that stage of the process.

It is clear to us that it can happen that some modules can't be switched, as they may lead to impossible combinations. For example, a quorum module that waits for some zones when the system has only one zone.

In the same way as the modules can be switched, the versioning mode can also be. A system that uses a simple integer to data versioning, it is possible to change other type, e.g., a vector clock. For this, a new versioning class implementation must implement the version interface of our framework.

4.6 Implementation experience

The choice of adapting existing systems led us to a more interative development process. Starting by the time needed to obtain sufficient knowledge of the system that we are modifying and ending with dealing some system design features that made it difficult to include the framework in the system.

The complexity and size of the code were one of the main adversities we faced when we were modifying the Voldemort system (4.3.2). In order to be prepared to modify the system, it was necessary to obtain a deep internal knowledge of all system mechanisms related to consistency. Out of the box, Voldemort offers a lot of customization options and code optimizations so, sometimes the code wasn't easy to understand. In addition, sometimes the available documentation is not up to date with all new features or design choices. Therefore, we anticipated this type of difficulties.

Unlike the Voldemort system, DKVF (4.3.1) was designed to be used by the academic community. As a result, the system code produced using DKVF is clean and simple. So, although while implementing COPS using DKVF we had the process of learning about the system, it was much smoother than with Voldemort.

Moving on from the framework implementation into the systems, this process of adapting existing systems led us to have to rethink the architecture initially proposed and redo it in some parts to make it more modular. It happened not only because the system we were modifying had scenarios that we hadn't thought of before, but also because some better alternatives were emerging due to the iterative process of framework implementation.

We were able to implement the framework in the systems described above, maintaining the consistency guarantees that the system originally had. However, changing these consistency guarantees is as simple as changing the modules that are being instantiated at startup, defined in the Configurations class.

To test this possibility of exchanging modules and consequently changing the consistency model, we tested and successfully managed to change the versioning from vector clock to Lamport clock in Project Voldemort, and from Lamport clock to vector clock in COPS with DKVF. We also exchanged the causal consistency of the COPS with DKVF system for eventual consistency, simply by changing the Delivery Conditions module for one with no dependencies awareness from an eventually consistent system.

Chapter 5

Evaluation

5.1 Methodology

To evaluate our proposal, we will compare the existing system implementation of both systems chosen against our equivalent implementation of the systems using our framework (4.3). To support this, we are going to use two metrics: latency and throughput, in order to measure the overhead between these two implementations.

To compare the latency and throughput we will execute the same operations in both system implementations, and measure the overhead of adding a new layer to the system.

5.2 DKVF

5.2.1 Experimental Setup

DKVF (4.3.1) includes a YCSB driver [57]. YCSB means Yahoo! Cloud Serving Benchmark which is a framework used as a tool for evaluating the performance of key-value stores. We used this already implemented feature of DKVF, making variations to the number of operations and percentages of reads and writes, to give us the throughput and latency. In order to evaluate this system, we built a cluster as shown in Figure 5.1 where we have three servers with three partitions each, running a data store. The following representation X_Y identifies the machines, in which X means replica number and Y partition number. Two of three replicas of the cluster are connected to a hub that is connected to three clients each. Each partition is connected to other replicas with the same partition. The remaining replica actuates in the system as another replication point, not being connected to any client. We are assuming full replication between replicas.

Each node in the cluster represented on figure 5.1 runs in an independent machine. For this, we used the INESC-ID [58] cluster. For the servers, we used a 1 vCPUs, 2.13

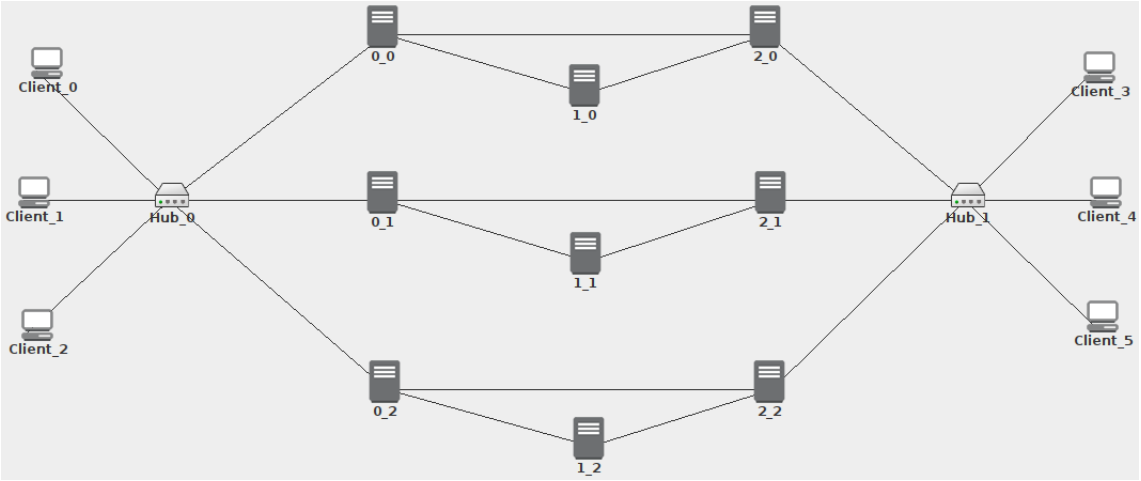

Figure 5.1: Experimental cluster representation

GHz, Intel Xeon E5506, 2 GiB memory RAM. For the clients, following the approach of the DKVF paper [52], we give more power to the clients to better utilize servers. We run clients on machines with 2 vCPU, 2.13 GHz, Intel Xeon E5506, 2 GiB memory RAM.

5.2.2 Experimental Results

In the implementation of COPS with DKVF, we perform the measurements varying the number of operations and the ratio of reads and writes operations on the YCSB properties. For this purpose, we chose 8 threads per client to increase the amount of load applied against the system. A *recordcount* (YCSB property) of 1000, which means that it will create 1000 records on load phase of the YCSB execution, and we vary the read:write operations ratio between 50:50 and 95:05, which correspond to a update heavy workload and a read-mostly workload. However, since we are just measuring the impact of our framework on both systems, we just want to have the same conditions on the original version and version with the framework to compare.

We started by doing a measurement for 50:50 operations ratio. The results are represented in Tables 5.1 and 5.2.

Operations Count	Throughput (ops/sec)	Write Latency (ms)	Read Latency (ms)
50000	1336,5	6943,4	4526,0
100000	1552,7	5913,7	4095,0
200000	1775,9	5241,4	3648,0
300000	1927,0	4792,5	3403,1
400000	1987,3	4602,5	3368,0

Table 5.1: COPS with DKVF original implementation - 50:50 operations ratio

Operations Count	Throughput (ops/sec)	Write Latency (ms)	Read Latency (ms)
50000	1133,4	8009,8	5590,6
100000	1261,0	7154,1	5207,1
200000	1499,9	6134,5	4350,0
300000	1490,5	6186,5	4414,3
400000	1449,9	6349,5	4450,0

Table 5.2: Modified COPS with DKVF - 50:50 operations ratio

By default, the execution of YCSB gives us a result per client. In this case, we used 6 clients, each 3 against a different replica of the system. To help better understand our results, we present in these both tables an average of all the clients results.

Table 5.3 shows the impact of our framework on the system. The overhead number is between 15% and 23%.

Operations Count	Throughput Overhead	Write Latency Overhead	Read Latency Overhead
50000	15,2%	13,3%	19,0%
100000	18,8%	17,3%	21,4%
200000	18,4%	17,5%	18,0%
300000	22,7%	22,5%	22,9%
400000	23,1%	22,1%	21,4%

Table 5.3: COPS with DKVF original vs modified overhead - 50:50 operations ratio

We did the same experiment with all the same conditions except the operations ratio that we fixed on 95:05. The results shown in Tables 5.4, 5.5 and the consolidated overhead on Table 5.6 are close to the previous experiment.

We made more variations to the parameters and reran the experiments. We found out that the overhead was always close to the numbers of the two previous experiments presented. So we believe that this is approximately the real overhead value of our solution. There are reasons behind these numbers that we will discuss in section 5.4.

Operations Count	Throughput (ops/sec)	Write Latency (ms)	Read Latency (ms)
50000	1850,5	12812,8	3610,3
100000	1977,5	12767,5	3443,9
200000	2210,9	10561,1	3185,4
300000	2291,4	10580,2	3059,3
400000	2374,7	9392,1	3027,3

Table 5.4: COPS with DKVF original implementation - 95:05 operations ratio

Operations Count	Throughput (ops/sec)	Write Latency (ms)	Read Latency (ms)
50000	1487,4	13944,9	4576,4
100000	1795,9	12229,7	4075,6
200000	1866,0	10104,6	3897,6
300000	1959,9	12223,7	3588,3
400000	1948,7	10058,0	3752,8

Table 5.5: Modified COPS with DKVF - 95:05 operations ratio

Operations Count	Throughput Overhead	Write Latency Overhead	Read Latency Overhead
50000	19,6%	8,1%	21,1%
100000	9,2%	-4,4%	15,5%
200000	15,6%	-4,5%	18,3%
300000	14,5%	13,5%	14,7%
400000	17,9%	6,6%	19,3%

Table 5.6: COPS with DKVF original vs modified overhead - 95:05 operations ratio

5.3 Project Voldemort

5.3.1 Experimental Setup

This system's source code also includes a benchmark tool like DKVF. However, it doesn't allow us to test the modifications that we did because it is a pure storage engine test. So

this is useful to test and compare new storage engines with the system but not useful for our context.

To measure the impact of our framework on this system, we executed different sequences of operations and measured the time between the begin and the end.

For this experiment, we built an experimental enviroment with two pairs of three nodes located in different networks and two clients that send queries to the clusters. All of them run on independent machines with the following configurations: 4 vCPUs, 2.13 GHz, Intel Xeon E5506, 4 GiB memory RAM using machines at INESC-ID [58] cluster.

5.3.2 Experimental Results

In the Voldemort system, we did a similar work to the evaluation of DKVF. Altough we didn't use YCSB to evaluate this system, we create workloads that simulate the same scenarios.

Using the same workloads in both versions and varying the number of operations for two different read:write operations ratio, we measured the overhead of our framework into the system.

Tables 5.7 and 5.8 shows the throughput and the overhead calculated by the difference between throughput of the original implementation and the modified version with our framework.

Operations Count	Throughput Framework (ops / sec)	Throughput Original (ops / sec)	Overhead (%)
50000	3421,4	3809,2	10,2%
100000	4548,6	5005,5	9,1%
200000	4846,9	5251,4	7,7%
300000	4911,0	5349,0	8,2%
400000	4818,2	5190,2	7,2%

Table 5.7: Project Voldemort - 50:50 operations ratio

Operations Count	Throughput Framework (ops / sec)	Throughput Original (ops / sec)	Overhead (%)
50000	5644,6	6309,2	10,5%
100000	6852,1	7457,1	8,1%
200000	7691,7	8161,6	5,8%
300000	7685,0	8471,2	9,3%
400000	7745,3	8716,9	11,2%

Table 5.8: Project Voldemort - 95:05 operations ratio

5.4 Discussion

The results of both systems show that our solution adds an overhead to the system. There are reasons for the values obtained that we will describe with an individual analysis of the values obtained from each system.

Starting with COPS implemented with DKVF, the overhead for the 50:50 operations ratio (Table 5.3) varies between 15% and 23%. While for the 95:05 the overhead values (Table 5.6) are approximately the same values if we exclude write latency from this comparision.

Looking at the 50:50 operations ratio tables (5.1 and 5.2) and doing a comparison against the 95:05 operations ratio tables (5.4 and 5.5), an increase of almost double the write latency of 50:50 results is noticeable when compared with 95:05. COPS offers a causal+ consistency that deals with dependencies between operations. The 95:05 write latency is almost double the one of the 50:50 operations ratio because with such increase of reads, before the writes, it creates a bigger dependencies list that needs to be satisfied before the write operation can be considered complete. However, in the 95:05 operations ratio (Table 5.6), we have two cases of write latency where the system with our framework has better performance (negative percentages). However, given that for the 50:50 operations ratio (table 5.3) the values are more consistent and due to the low quantity of write operations (only 5%), these are not values to which we have given relevance.

Let's now focus on analyzing the reasons behind a general throughput drop in the system for both operations ratio. As previously stated, there is a throughput overhead in the modified version, with our framework, when compared with the original implementation system. We decided to investigate the real impact of the type conversion between our framework and the system's, given that, in order to be possible to have modularity, we defined some types that messages that arrive at the system must be converted and then reconverted again to the original format when leaving the framework. For this, we executed the same operation with a list of previously created dependencies and we measured the time spent on type conversion and the total time of execution. The total time measured for this operation was 3.8 ms on the original implementation, and 4.9 ms on our implementation with our framework. However, from these 4.9 ms, we measured a type conversions time of 0.8ms. These results led us to conclude that most of the overhead of our framework is due to type conversion.

On Project Voldemort system, we were only able to measure the throughput due to the tool that we chose for these measurements. However, it gave us an estimate of the impact of our framework. Table 5.7 for 50:50 operations ratio and Table 5.8 for 95:05 operations ratio gave us similiar results to those obtained from the DKVF system. Also in this system most of the overhead is caused by type conversions.

Chapter 6

Conclusion

Distributed replicated systems tend to be built with a consistency model implemented coupled with their implementation, making switching between consistency models difficult. Thus, usually when a consistency model of a system has to be changed, either the system code needs to be deeply rewritten or replaced by a different consistency system.

Our analysis to existent systems, found that there are components of different consistency models that even although they can define a different semantics or use different mechanisms to provide consistency to the system, they serve the same purpose with a similar base approach.

We proposed a modular abstraction to the consistency model and a respective framework which makes use of that abstraction to extract the consistency implementation to a layer that can be implemented in a modular isolated manner. We found that this abstraction fits into all of the analyzed consistency protocol implementations and our architecture allows the developers to change the consistency model of a system at build time.

To evaluate our proposal, we implemented two different systems into our framework: COPS using DKVF, and Project Voldemort. We measured the throughput and associated overhead between original implementation and modificated implementation with our framework. Although the measured values show an impact of our framework for an increase of the system flexibility, our analysis revealed that a large part of this impact is due to the conversion of data type in our framework. These are values that could be further optimized.

6.1 Future Work

We have focused on identifying modular replaceable components in replicated systems. Our experience suggests that this approach may be extended to the transaction support and management of distributed transactional systems. Eventually both modular frameworks

could be integrated in the same model.

In a next version of the framework presented, the data types conversion should be revisitated in order to improve the performance and consequently reduce the overhead.

Finally, it would be interesting if in a future work it was possible to exchange some framework modules at runtime instead of just at build time.

Bibliography

[1] Will Wang. *How Microservices Saved the Internet.* 2018. URL: https://hackernoon.com/how-microservices-saved-the-internet-30cd4b9c6230 (visited on 01/21/2019).

[2] Google. *Zeitgeist 2012 – Google.* 2012. URL: https://archive.google.com/zeitgeist/2012/ (visited on 05/17/2019).

[3] T. Simonite. *Moore's law is dead. Now what?* 2016. URL: https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/ (visited on 01/21/2019).

[4] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud." In: *2015 10th Colombian Computing Conference, 10CCC 2015.* 2015, pp. 583–590. ISBN: 9781467394642. DOI: 10.1109/ColumbianCC.2015.7333476.

[5] K. Arsov. *What Are Microservices, Actually?* 2017. URL: https://dzone.com/articles/what-are-microservices-actually (visited on 01/21/2019).

[6] R. Guerraoui, M. Pavlovic, and D. A. Seredinschi. "Incremental consistency guarantees for replicated objects." In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016.* 2016, pp. 169–184. ISBN: 9781931971331. arXiv: 1609.02434.

[7] J. Enough, D. Systems, T. Be, and T. Lipcon. "Design Patterns for Distributed Non-Relational Databases." In: *Cloudera* (2009). ISSN: 10959203. DOI: 10.1126/science.1095048.

[8] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. "Bolt-on causal consistency." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 2013, pp. 761–772. ISBN: 9781450320375. DOI: 10.1145/2463676.2465279.

[9] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. "Making geo-replicated systems fast as possible, consistent when necessary." In: *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012.* 2012, pp. 265–278. ISBN: 9781931971966.

[10] K. Ranganathan, A. Iamnitchi, and I. Foster. "Improving data availability through dynamic model-driven replication in large peer-to-peer communities." In: *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid 2002*. 2002. ISBN: 0769515827. DOI: 10.1109/CCGRID.2002.1017164.

[11] A. Silberstein, A. Silberstein, B. F. Cooper, B. F. Cooper, U. Srivastava, U. Srivastava, E. Vee, E. Vee, R. Yerneni, R. Yerneni, R. Ramakrishnan, and R. Ramakrishnan. "PNUTS: Yahoo!'s Hosted Data Serving PLatform." In: *SIGMOD* (2008). ISSN: 21508097. DOI: 10.1145/1376616.1376693.

[12] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." In: *Communications of the ACM* 21.7 (1978), pp. 558–565. ISSN: 15577317. DOI: 10.1145/359545.359563.

[13] F. B. Schneider. "Replication Management using the State Machine Approach." In: *ACM Computing Surveys* 22.4 (1990), pp. 299–319.

[14] X. Defago, A. Schiper, and N. Sergent. "Semi-passive replication." In: *Proceedings of the IEEE Symposium on Reliable Distributed Systems*. 1998, pp. 43–50. DOI: 10.1109/reldis.1998.740473.

[15] L. Rodriguez and M. Raynal. "Atomic broadcast in asynchronous crash-recovery distributed systems." In: *Proceedings - International Conference on Distributed Computing Systems*. 2000, pp. 288–295. DOI: 10.1109/icdcs.2000.840941.

[16] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. "The primary-backup approach." In: *Distributed systems (2nd Ed.)* (1993).

[17] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. "Primary-Backup Protocols: Lower Bounds and Optimal Implementations." In: 1993, pp. 321–343. DOI: 10.1007/978-3-7091-4009-3_14.

[18] R. Ladin, B. Liskov, and L. Shrira. "Lazy replication. Exploiting the semantics of distributed services." In: *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*. 1990, pp. 43–57. DOI: 10.1145/93385.93399.

[19] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. "Providing High Availability Using Lazy Replication." In: *ACM Transactions on Computer Systems (TOCS)* 10.4 (1992), pp. 360–391. ISSN: 15577333. DOI: 10.1145/138873.138877.

[20] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. "Epidemic algorithms for replicated database maintenance." In: *ACM SIGOPS Operating Systems Review* 22.1 (1988), pp. 8–32. ISSN: 0163-5980. DOI: 10.1145/43921.43922.

[21] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. "Partial replication in the Database State Machine." In: *Proceedings - IEEE International Symposium on Network Computing and Applications, NCA 2001*. 2001, pp. 298–309. ISBN: 0769514324. DOI: 10.1109/NCA.2001.962546.

[22] M. Shapiro, K. Bhargavan, Y. Chong, and Y. Hamadi. "A formalism for consistency and partial replication." In: *Microsoft Research* (2004). URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2004-58.pdf.

[23] E. a. Brewer and U. C. Berkeley. "Towards Robust Distributed System." In: *Networks* (2000). DOI: 10.1145/343477.343502.

[24] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." In: *ACM Sigact News* 33 (2002), pp. 51–59.

[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: *Sosp* (2011), pp. 1–16.

[26] *Microsoft Data Platform | Microsoft.* URL: https://www.microsoft.com/en-gb/sql-server/ (visited on 08/22/2020).

[27] *MySQL.* URL: https://www.mysql.com/ (visited on 08/22/2020).

[28] *PostgreSQL: The world's most advanced open source database.* URL: https://www.postgresql.org/ (visited on 08/22/2020).

[29] L. Lamport. "The Part-Time Parliament." In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169. ISSN: 07342071. DOI: 10.1145/279227.279229.

[30] A. Lakshman and P. Malik. "Cassandra - A decentralized structured storage system." In: *Operating Systems Review (ACM)* 44.2 (2010), pp. 35–40. ISSN: 01635980. DOI: 10.1145/1773912.1773922.

[31] Voldemort. *Developer Info - Voldemort.* 2020. URL: https://www.project-voldemort.com/voldemort/ (visited on 08/25/2020).

[32] 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* (2007). ISSN: 01635980. DOI: 10.1145/1323293.1294281.

[33] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. "Existential consistency: Measuring and understanding consistency at Facebook." In: *ACM Symposium on Operating Systems Principles* 15 (2015), pp. 295–310. DOI: 10.1145/2815400.2815426. URL: http://sigops.org/sosp/sosp15/current/2015-Monterey/printable/240-lu.pdf.

[34] W. Vogels. "Eventually consistent." In: *Communications of the ACM* (2009). ISSN: 00010782. DOI: 10.1145/2576794.

[35] *Consistency Models.* URL: https://jepsen.io/consistency (visited on 05/19/2019).

[36] M. P. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492. ISSN: 15584593. DOI: 10.1145/78969.78972.

[37] I. Zhang. *Operation Ordering in Systems*. URL: https://irenezhang.net/research/consistency.html (visited on 05/19/2019).

[38] L. Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs." In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 690–691. ISSN: 00189340. DOI: 10.1109/TC.1979.1675439.

[39] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. "Managing update conflicts in Bayou, a weakly connected replicated storage system." In: *ACM SIGOPS Operating Systems Review* 29.5 (1995), pp. 172–182. ISSN: 0163-5980. DOI: 10.1145/224057.224070.

[40] M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. "PRACTI Replication for Large-Scale Systems." In: *Technical Report: UT Austin* (2006). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.58.1985{\&}rank=1.

[41] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. "Session guarantees for weakly consistent replicated data." In: *Parallel and Distributed Information Systems - Proceedings of the International Conference*. 1994, pp. 140–149. DOI: 10.1109/pdis.1994.331722.

[42] P. Viotti and M. Vukolić. "Consistency in non-transactional distributed storage systems." In: *ACM Computing Surveys* 49.1 (2016). ISSN: 15577341. DOI: 10.1145/2926965. arXiv: 1512.00168. URL: http://arxiv.org/abs/1512.00168.

[43] F. Freitas, J. Leitão, N. Preguiça, and R. Rodrigues. "Fine-grained consistency upgrades for online services." In: *Proceedings of the IEEE Symposium on Reliable Distributed Systems*. Vol. 2017-Septe. 2017, pp. 1–10. ISBN: 9781538616796. DOI: 10.1109/SRDS.2017.9.

[44] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web." In: *Proc. of ACM Symposium on Theory of Computing (STOC)* (1997). ISSN: 0012821X. DOI: doi:10.1145/258533.258660.

[45] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. "Spanner: Google's Globally Distributed Database." In: *ACM Trans. Comput. Syst.* (2012). ISSN: 07342071. DOI: 10.1145/2491245.

[46] E. Brewer. "Spanner, TrueTime & The CAP Theorem." In: *Google White Papers* (2017).

[47] B. W. Lampson, B. W. Lampson, D. Lomet, and D. Lomet. "A New Presumed Commit Optimization for Two Phase Commit." In: *19th VLDB Conference*. Vol. 927. 1993, pp. 1–9. ISBN: 1-55860-152-X.

[48] A. Colyer. *Bolt-on Causal Consistency - the morning paper*. URL: https://blog.acolyer.org/2015/09/01/bolt-on-causal-consistency/.

[49] V. Balegas, C. Li, M. Najafzadeh, D. Porto, A. Clement, S. Duarte, C. Ferreira, J. Gehrke, J. Leitão, N. Preguiça, R. Rodrigues, M. Shapiro, and V. Vafeiadis. *Geo-Replication: Fast If Possible, Consistent if Necessary*. Tech. rep. 2016, pp. 81–92.

[50] J. Du, A. Roy, W. Zwaenepoel, and C. Iorgulescu. "GentleRain : Cheap and Scalable Causal Consistency with Physical Clocks." In: *SOCC '14 Proceedings of the ACM Symposium on Cloud Computing* (2014). DOI: 10.1145/2670979.2670983.

[51] P. Carbonnelle. *PYPL PopularitY of Programming Language index*. 2020. URL: http://pypl.github.io/PYPL.html (visited on 08/28/2020).

[52] M. Roohitavaf and S. Kulkarni. "DKVF: A framework for rapid prototyping and evaluating distributed key-value stores." In: *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 912–915. ISBN: 9781450359375. DOI: 10.1145/3238147.3240476. arXiv: 1801.05064.

[53] Google. *Protocol Buffers | Google Developers*. 2020. URL: https://developers.google.com/protocol-buffers/ (visited on 08/25/2020).

[54] M. Roohitavaf. *roohitavaf/DKVF*. 2016. URL: https://github.com/roohitavaf/DKVF (visited on 08/28/2020).

[55] LinkedIn. *LinkedIn*. URL: https://www.linkedin.com/ (visited on 08/28/2020).

[56] X. Yingzhong. *Enable server side routing strategy in Java client · Issue #112 · voldemort/voldemort*. 2013. URL: https://github.com/voldemort/voldemort/issues/112 (visited on 08/28/2020).

[57] 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, SoCC '10*. 2010, pp. 143–154. ISBN: 9781450300346. DOI: 10.1145/1807128.1807152.

[58] INESC-ID. *INESC-ID*. 2020. URL: https://www.inesc-id.pt/ (visited on 09/04/2020).