David Canilhas Romão

Degree in Computer Science and Engineering

Dynamic Data Placement in Cloud/Edge Environments

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: João Leitão, Assistant Professor,
NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2021

This work was partially supported by FC&T through grants NOVA
LINCS (UID/CEC/04516/2013) and project NG-STORAGE (PTDC/CC-INF/32038/2017).

Abstract

Nowadays the Cloud Computing model is the most widely used paradigm to support the operation of large scale distributed systems. However, this model has its own limitations, such as the limited bandwidth to access the cloud data centres. To circumvent these limitations the Edge Computing model has emerged and been rising in popularity recently, mainly due to the growth of the Internet of Things (IoT) and applications that generate significant amounts of data. We aim to enable applications that currently use the Cloud Computing model to be able to take advantage of the Edge.

Most applications will not fit well in a model where they live solely on the Edge, as they still need guarantees offered by data centres, namely durability of application state. Ideally we would want to take advantage of both models. This however, raises the question on how to create a system that integrates both cloud and edge to take the best of both worlds. As the edge is composed of a multitude of devices, each with different characteristics where the computational resources available may be dynamic, it presents a very different environment when compared to the one commonly seen in data centres where everything is highly controlled.

In this work we will focus on a particular key aspect of any modern distributed system, the data management layer. In particular we aim at creating a novel distributed data store that dynamically stores application (and user) data both in the cloud infrastructure and several edge devices. Furthermore we want to explore and devise mechanisms to dynamically manage the data replicated across different edge locations and explore replication protocols offering different consistency guarantees, such as eventual consistency and causal consistency.

Keywords: Distributed datastore systems, Edge Computing, automated data placement, causal+ consistency, genuine partial replication.

Resumo

Hoje em dia o paradigma de Computação na Nuvem é o mais utilizado para suportar sistemas distribuídos de grande escala. No entanto este paradigma tem as suas limitações, nomeadamente o sobrecarregamento da rede com o tráfego necessário para aceder aos centros de dados na Nuvem. De forma a ultrapassar estas limitações o paradigma de Computação na Berma apareceu e tem vindo a ganhar popularidade recentemente, principalmente devido ao crescimento da Internet das Coisas e ao aparecimento de aplicações que geram um quantidade significativa de dados. O objetivo principal do trabalho é que as aplicações que de momento utilizam o modelo de Computação na Nuvem sejam capazes de tirar partido da Computação na Berma.

A maioria das aplicações não se adapta bem a um paradigma onde os seus recursos apenas se encontram na Berma, devido a dependerem de propriedades que são mais simples de obter dentro de um centro de dados, nomeadamente durabilidade do estado da aplicação. Idealmente seria possível tirar partido de ambos os paradigmas, o que leva ao problema de como criar um sistema que integre a Nuvem e a Berma e aproveite o melhor dos dois. Devido à Berma ser composta por uma grande variedade de dispositivos, cada um com caraterísticas diferentes e os recursos computacionais disponíveis podem ser dinâmicos, o que leva a que o ambiente na Berma seja consideravelmente diferente quando comparado com a Nuvem, que por norma se trata de um ambiente altamente controlado.

No trabalho apresentado neste documento iremos focar num ponto particular para qualquer sistema distribuído moderno, a camada de armazenamento. Em particular o objetivo do nosso trabalho é criar um novo sistema de armazenamento distribuído que suporte armazenar os dados das aplicações (e dos utilizadores) de forma dinâmica na infraestrutura na Nuvem tal como em vários dispositivos na Berma. Ainda mais, pretendemos explorar e desenvolver mecanismos para gerir os dados replicados nos dispositivos na Berma de forma dinâmica tal como explorar protocolos de replicação que ofereçam vários níveis de consistência (ex. eventual, causal+)

Palavras-chave: Sistemas de armazenamento distribuídos, computação na Berma, posicionamento de dados automatizado, consistência causal+, replicação parcial genuína.

Contents

List of Figures xi

Acronyms xiii

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Problem definition . . . 2
- 1.4 Contributions . . . 3
- 1.5 Document organisation . . . 3

2 Related work 5
- 2.1 Edge Computing . . . 5
- 2.2 Replication . . . 6
  - 2.2.1 Replication Schemes . . . 7
  - 2.2.2 Replication Strategies . . . 8
  - 2.2.3 Replication for Edge Computing . . . 9
- 2.3 Cache . . . 9
  - 2.3.1 Memcached . . . 10
  - 2.3.2 Discussion . . . 11
- 2.4 Consistency on Distributed Systems . . . 12
  - 2.4.1 Strong Consistency . . . 13
  - 2.4.2 Weak Consistency . . . 13
  - 2.4.3 Tracking Causal Dependencies . . . 14
  - 2.4.4 Consistency for Edge Computing . . . 16
- 2.5 Automated Location-Aware Data Placement . . . 16
  - 2.5.1 Offline Data Placement . . . 18
  - 2.5.2 Online Data Placement . . . 19
- 2.6 Distributed Databases . . . 20
  - 2.6.1 Dynamo . . . 21
  - 2.6.2 Cassandra . . . 23
  - 2.6.3 COPS . . . 24
  - 2.6.4 ChainReaction . . . 25

2.6.5 Redis . . . 26
2.7 Summary . . . 28

3 Causal consistency with dynamic data placement 29
3.1 Overview . . . 30
3.2 System Model . . . 30
3.3 Overlay Structure . . . 31
3.4 Dynamic data placement . . . 34
3.4.1 Creating new key replicas on edge nodes . . . 34
3.4.2 Removing replicas on edge nodes . . . 35
3.5 Causality . . . 35
3.5.1 Adapted Version Vectors . . . 36
3.5.2 Detecting concurrent operations . . . 36
3.6 Replica Migration . . . 39
3.7 Client . . . 40
3.7.1 Metadata . . . 40
3.8 Summary . . . 42

4 Implementation 43
4.1 Datastore selection . . . 43
4.2 Redis Internals . . . 43
4.2.1 Event Loop . . . 44
4.2.2 Command Execution . . . 44
4.2.3 Replication and Persistence . . . 45
4.2.4 Key Expiration . . . 45
4.2.5 Eviction of keys . . . 45
4.3 Prototype Implementation . . . 45
4.3.1 Communication Links . . . 46
4.3.2 Command List Serialization . . . 46
4.3.3 Optimizing conflict detection . . . 47
4.3.4 Dynamic Data Placement . . . 47
4.3.5 Replication . . . 48
4.3.6 Autonomous replicas removal on edge nodes . . . 49
4.3.7 Client . . . 50
4.4 Summary . . . 51

5 Evaluation 53
5.1 Experimental Setup . . . 53
5.2 Experimental parameters . . . 55
5.2.1 Baseline . . . 56
5.3 Results . . . 56

5.3.1 Multiple configurations and the impact of various parameters in the system . . . 56
5.3.2 Baseline . . . 59
5.4 Summary . . . 61

6 Conclusion and future work **63**
6.1 Conclusion . . . 63
6.2 Future Work . . . 64

Bibliography **65**

List of Figures

3.1 Overlay 32

5.1 Performance impact for multiple acks configurations, with 2500 clients and MD=1 57
5.2 Performance impact for multiple Metadata Distance configurations, with acks=1 59
5.3 (B)aseline vs (E)dge 60

Acronyms

AOF	Append-Only File.
CDN	Content Distribution Network.
DHT	Distributed Hash Table.
FIFO	First In First Out.
IoT	Internet of Things.
LFU	Least Frequently Used.
LRU	Least Recently Used.
QoS	Quality of Service.
WAN	Wide Area Network.
YCSB	Yahoo! Cloud System Benchmark.

Chapter 1

Introduction

1.1 Context

The Cloud Computing paradigm appeared apparently 20 years ago, proposing to share the infrastructure between different applications to maximise resource usage and significantly reduce hosting costs. The infrastructure is maintained and managed by a provider, offering an abstraction layer that enables to adjust available resources to the application's needs. This layer started by being capable of handling automatic scaling and fault tolerance but have now evolved to offer a plethora of functionalities and services ready to be used by the applications. The appearance of this new model significantly simplified and accelerated the development of new services, since applications were built on top of this abstract layer that already offered strong guarantees needed by all applications. Previous to the appearance of the Cloud, there was an upfront cost to build an infrastructure for a service without having any idea of its (future) popularity, possibly leading to two undesired scenarios. Over provisioned resources would lead to unused resources due to the demand being lower than expected, while the popularity is still growing or not reaching the desired level. Failing to provision adequately, might lead the infrastructure to be incapable of handling an unexpected surge in demand. The Cloud avoided these scenarios by offering a "pay as you go"pricing for their services that can scale as needed according to the application's necessities. Allowing to keep a low cost for the infrastructure, or even completely free nowadays, during development/initial stages of an application life cycle.

This model works well enough with the typical web application where the user interaction is very limited, the data flow is mostly uni-directional (server to client), with an occasional submission by the client. However, it also presents some limitations, the rise of Internet of Things (IoT) devices is causing an exponential growth on the rate at which data is generated, processing all this data at a data centre is not feasible for two reasons,

data many times is short-lived, meaning that is is only relevant for a small period of time and needs to be processed before it becomes irrelevant, and because, as the volume of data grows, it will reach a point where it will not be possible to send all to the data centre due to the bandwidth limitations of the network links and the processing power required to do so.

This has caused the appearance of Edge Computing, a new paradigm where some tasks can be moved to devices outside of data centres, which distributes the load of the system, reduces latency; and, importantly, reduces the amount of traffic reaching the data centre by pre-processing the data. These devices usually do not have a lot of processing power and storage, being only capable of executing simple tasks that are not resource intensive.

The key insight is that some computations and data management tasks can be moved to the edge of the system (and beyond cloud data centres perimeters) where data is effectively produced and can also be processed.

1.2 Motivation

Edge Computing is still quite new and its capabilities have not been fully explored yet. It aims at solving the new challenges that come with the growth of the IoT as well as promising extreme low latency, allowing the creation of applications that rely heavily on user interaction with the service. To meet the requirements of such applications, various challenges must be addressed. To offer low latency, data should be stored in locations close to those where it is (mostly) accessed, either in nearby data centres or on the edge. When making decisions about where data should be placed, various factors must be taken into account, mainly the limited storage and computing power available on the Edge devices. Some research efforts already exists on data placement for geo-distributed systems, such as [1, 4, 17, 18, 31–33, 35], however, they do not consider placing data on the Edge, most only offer static locations which are not capable of autonomously adapting to changes in usage/access patterns patterns, which is critical when we are limited by the amount of resources available on the edge and do not want to waste storage space with data that is no longer required.

1.3 Problem definition

The primary goal of this work is to explore how the cloud and edge computing paradigms can be integrated in order to improve performance of distributed datastore systems. To do so we defined the following two main problems that were be addressed in this work :

1. Creating a data placement scheme that is highly dynamic, capable of creating and removing data replicas as required based on the locations where data is accessed,

the frequency of those accesses and the available computational resources, with a focus on keeping the costs down and an acceptable low latency.

2. A replication protocol with dynamic membership that enables adding and removing data replicas as required, with support for partial replication in order to increase its (space) efficiency. It must keep the replicas up-to-date, handling conflicts of concurrent operations in a way that ensures causal consistency. As nodes on the Edge are subject to a higher chance of failure than on the Cloud so the protocol has to be capable of quickly reacting to failures as well as ensuring data durability on the Edge,

1.4 Contributions

The main contributions presented in this thesis are the following:

1. A novel protocol for data replication which ensures causal+ consistency with support for partial replication that fits well in the environment created by the integration of the Edge and Cloud computing paradigms.

2. A online data placement scheme that integrates the Edge with the Cloud, with a main focus on being highly dynamic and capable of adapting to the available computational resources and access patterns, without compromising on causal+ consistency and durability offered at the replication level.

3. A implementation of both the replication protocol and the data placement scheme over Redis[1], a popular open source datastore.

1.5 Document organisation

The document is organised as follows :

Chapter 2 introduces the concept of Edge computing and discuss various areas of work relevant to our main contribution, present other works in those areas and discuss why they are relevant to us.

Chapter 3 defines the system model and details the main components of our solution, followed by the description of the algorithms proposed and the reasoning behind them.

Chapter 4 explains why Redis was used as the base for our implementation of the algorithms defined in Chapter 3, followed by an explanation of relevant internals of Redis, ending with a presentation and discussion of relevant implementation details.

[1] https://redis.io/

Chapter 5 evaluates our implementation, compares it with the base system and analyses the results.

Chapter 6 presents the conclusions and possible future work.

Chapter 2

Related work

In this chapter, we will discuss and introduce various relevant concepts considering the goal of the work.

In Section 2.1 we focus on Edge Computing, as we want to build a system that will live on both the Edge and the Cloud, we need to understand the challenges that come with it.

In Section 2.2 we introduce and discuss various replications techniques, which are used to ensure fault tolerance. This has been widely studied and many techniques have been developed, which will be introduced and discussed.

In Section 2.3 we introduce the concept of cache, why it is useful for various applications and discuss which aspects overlap with the goal of our work that might be useful.

In Section 2.4 we introduce various consistency models, discuss their limitations and trade-offs that have been made to build large scale systems efficiently.

In Section 2.5 we focus on the need for geographically distributed systems to store the data on the best location to save costs and improve performance. We discuss the impact on performance that placing data on ideal locations can have, as opposed to typical systems that just focus on load balancing and use consistent hashing to partition the data. We also introduce some proposed solutions and discuss their limitations.

In Section 2.6 we introduce existing distributed databases, discuss their main limitations and the advances that have been made recently, concerning the features they offer and the guarantees they provide.

2.1 Edge Computing

This new computing paradigm comes with various challenges that must be solved to achieve its full potential. Edge Computing includes any computation that can be executed

outside a data centre and an Edge Device can be any of the existing devices between the end-user and the data centre, that is IoT devices, smartphones, routers, and servers located at ISPs among others[22]. This means that Edge Computing is a very broad area to be explored that can be split into various levels, depending on how further away from the data centre we are, that can be used for different purposes. It is not yet clear what purpose each level will fulfil.

Most applications will not depend solely on Edge Computing, requiring some sort of integration and coordination with data centres. A service-based approach is proposed in [34], which splits the processing between the Edge and the Cloud. The raw data stream from sensors is converted into high-level events on the edge, reducing upload traffic to the Cloud since data does not have to be constantly streamed any longer. These events are then scheduled for processing on the Cloud, considering available resources.

Edge devices have a primary function they must fulfil, but most of the time the resources are not being fully used, leaving space to run other processes. These unused resources can be taken advantage by other applications, this requires knowing the application's resource requirements and ensuring it does not consume a critical amount, that is, more resources than the ones unused by the primary function of the Edge device. As in the Cloud model, scaling might be required to adjust to demand and maintain the desired Quality of Service (QoS). In [30] a framework to manage resources on the Edge is proposed, capable of auto-scaling applications based on their priority and resource usage. This is ensured by a periodic timer that compares the actual latency to the desired one, accordingly adjusting increasing the application's resources, terminating others with an inferior priority if required. When the available resources are not enough to improve the latency, the application is migrated back to the Cloud. Nodes are autonomously deleted if it is detected that they are not providing a better service when compared to the Cloud, that might happen if the edge node is overloaded leaving no unused resources. Applications are partially deployed to a device, that is,a control component executing in the Cloud chooses which users that node will serve and the relevant data is copied to the device, this data is then maintained locally. Users always start by connecting to the Cloud, where they are then redirected to an edge node if they have been assigned to a node. Operations on the Edge Device are done locally and then propagated to the Cloud in the background. When a replica is terminated all the data is synced with the Cloud before deletion. A limitation of this scheme is that it does not guarantee data durability in the case where a replica fails before propagating the changes to the data centre and does not recover. The changes that were not synchronized previously become lost permanently.

2.2 Replication

Nowadays replication is more important than it has ever been for most applications, due to the popularity level an application can reach worldwide and the requirements of critical applications. The Cloud model simplified the process of deploying an application

that makes use of replication by handling the process internally, all the developer needs to do is configure the system. It is still important to understand the process to decide the most adequate strategy. There are two main properties of replication that make it crucial:

1. High Availability and Fault Tolerance - When a replica fails for any reason, the system can still behave as normal (up to a certain amount of concurrent failures), increasing system availability. It is also important that no data is lost in case of a failure, this problem is also mitigated by the creation of redundant copies of the data.

2. Low Latency and Scaling - By replicating the data across different replicas, the load can be distributed evenly, possibly improving the response time. Since the data can be served from various replicas, horizontal scaling becomes possible, which is extremely important to make scaling sustainable.

Replication increases the system complexity, it will most likely hurt the performance of the applications under some circumstances while improving it in others. The replication scheme and protocol used must be carefully chosen based on the application's requirements.

2.2.1 Replication Schemes

When replicating data it can be placed across servers in the following two ways:

Total Replication: All replicas need to keep a copy of the whole data. This may increase the storage cost considerably and limits scalability, but simplifies the system complexity because (all) data can be accessed from any replica.

Partial Replication: Each replica only needs to keep a copy of a subset of the whole data. This works well with geographically distributed systems, where only the data that is required by nearby users needs to be kept on the replica, instead of the data from all the users worldwide, reducing storage costs significantly. This requires that we know which data each user will access and decide the best place to store the data, in Section 2.5 we will go into details on the challenges associated with such a solution.

A good use case for partial replication is social networks because users mostly access data from users and pages from their region. Consider Europe, each replica would contain data only from users of the country where it is located, saving on synchronization and storage costs.

Replication requires that metadata is propagated among the replicas, it is simpler to implement partial replication with replicas receiving metadata about all the data. However, this results in additional consumption of resources like bandwidth and processing power to process information about data that is not stored on the replica. Genuine Partial

Replication [14] is a specialization of partial replication, where replicas are only aware of data for which they are responsible, solving the problem mentioned previously, as fewer resources are used it enables better scalability. This strategy creates some complications when handling transactions with data present at more than one replica.

An example of partial replication used widely is cache, which provides only read operations similar to a read-only replica. It provides a faster response time for frequently accessed data, usually done on the client-side to reduce server load, although It can also be done on a Content Distribution Network (CDN) that executes on infrastructure nodes (but not limited to cloud infrastructures).

2.2.2 Replication Strategies

In this section, we present various relevant strategies that can be used to replicate the data in a distributed environment.

Primary-Backup: All write operations are first executed on the primary and later propagated to other replicas. Secondary replicas may execute read operations to distribute the system load but if strong consistency is not used then these values might be outdated.

Chain Replication: This strategy is a variation of primary-backup. presented in [29], is based on primary backup that allows higher throughput and availability while providing linearizability. The nodes (i.e. replicas) are organized in chains, that is, they are linearly ordered forming a chain, where the first node is the head and the last is the tail. Write operations are executed atomically on the head node which terminates the operation and replies to the client, and the result is then propagated down the chain until the tail is reached, effectively replicating the data objects. Read operations are sent to the tail, which generates the reply locally and responds immediately. When a node fails it is removed from the chain. Failures are supported as long as N-1 nodes are not failing concurrently.

Multi Master: Any operation can be executed on any replica. Operations will be propagated between replicas in the background, they may need to be ordered and there might exist conflicts on concurrent operations, so a protocol to handle them is necessary. Implementing strong consistency in such a system where operations can be executed in any replica without synchronizing all the replicas is hard to implement, for this reason this strategy is commonly used with weaker consistency models, which require less synchronization between replicas allowing for better scalability, there are however systems that implement in strong consistency which use a system to coordinate the operations, such as Paxos [21].

Quorum Replication: A quorum is a set of replicas that must execute an operation before it can be confirmed as complete. Typically there is a write quorum, W, and a read

quorum, R. Their sizes can vary as long as W + R > N, where N is the total number of replicas in the system. This ensures that there is at least one replica which contains the most recent state in both quorums. For operations that are not commutative, a total order is required when using quorums as to ensure that all replicas reach the same state.

2.2.3 Replication for Edge Computing

Partial replication is particularly relevant for Edge Computing, as it allows keeping only relevant data on the Edge. Genuine Partial Replication would be ideal due to its better performance and the fact that many devices on the Edge have limited resources so achieving good efficiency is desirable. The idea of moving data closer to the clients is not new, currently, services use CDN to store static files which are accessed frequently closer to the clients, like a YouTube video that is trending in a country or region. However, the problem becomes more complex when we want to deal with non-static data, and specially if this data can also be changed directly on the Edge and later propagated to the data centre, which is where our work will focus on.

The number of nodes in a system is already considerably large in data centres, with the move to the Edge that number is expected to grow even more. Hence our goal is to explore ways in which we can take advantage of having all these new nodes to improve the performance of the systems and the user experience. In order to take advantage of these new nodes an adequate replication protocol is required. The previously mentioned primary backup strategy is very limited in terms of scalability, the performance of the primary server is a bottleneck as operations can not be distributed through various servers. A multi-master or quorum system is more adequate, operations can be distributed through various servers, increasing performance with the growth in the number of replicas. Quorum systems have the advantage of being capable of adapting to the systems needs, the quorum sizes can be changed according to the expected ratio of write to read operations.

2.3 Cache

In this section, we will introduce the concept of caching, the Memcached system and possible optimizations to improve performance in a distributed setting.

A cache is a storage layer in the application architecture that is used to improve overall performance. It can be used to store frequently accessed content or computations results (i.e., databases queries). Browsers use this layer to locally store static content such as videos, images and web pages to improve the loading speed of content and save on bandwidth. Services can also introduce a caching on various levels inside their architecture. If placed between the application server and the database it can be used to temporarily store data, reducing the load on the database, inherently reducing response times (assuming the cache to be in-memory only). It can also be placed before the application server, where it can be used to store complex and frequent requests, improving the response time, and

again saving costs as complex operations can lead to many accesses and queries to be processed by database systems. Lastly, it can be used to store static content closer to the clients. There are CDN which are composed of various servers distributed geographically in strategic locations closer to the users, their sole purpose is to serve static content. It is widely used by video streaming services such as YouTube and Netflix to distribute the load on their systems as well as improve user experience by reducing transfer times.

2.3.1 Memcached

Memcached [10] is a high-performant caching solution that aims to make better use of available memory in a distributed system. It maintains an in-memory key-value store, capable of storing small blobs [1] , hence complex data structures must be pre-serialized. The system is composed of servers and a client side library. Servers maintain an in-memory key-value store, partitioned using a hashing algorithm. Clients can map keys to servers as they are aware of all the servers in the system. A particularity of this system is that servers are independent, there is no communication between servers. This means that synchronization is not required and replication is not supported. However, this allows for great scalability and simpler deployment, required to achieve the desired high performance.

Items have to eventually be removed from memory for two reasons :

- The system is running out of memory, here it relies on the Least Recently Used (LRU) [11] strategy to evict items from the key-value store.
- Items have to be invalidated to avoid returning stale data to the user, this is done when a client overwrites the value, deletes the value or when that value expiry date is reached. Each key has a corresponding expiry date that is changed with every write operation (to mitigate the possibility for a client to access a stale value).

2.3.1.1 Handling Failures

Fault tolerance for cache systems is a slightly different challenge when compared to persistent data storage systems. Since a cache is not the true source of the data, it is not necessary to ensure durability of the data stored. Hence it is not a problem when the data in a replica is lost due to a failure, it can be gradually filled again by the clients when they experience a cache miss and direct the request to the lower layer. Due to this property, a simple and correct solution would be to have clients redirect requests directly to the lower layer while the cache replica if the value is not available in any cache server, the problem with this is the performance degrades significantly due to the amount of requests that are now directed to the lower layer. This is the default behaviour intended for Memcached.

[1] A blob is a collection of binary data, typically used to store images or audio, but can be used to store anything

One solution that is typically used in distributed storage systems is redirecting requests to a failover server while a failure is occurring. This strategy makes sense for these systems because it ensures that requests can always be completed. However, this is not the case for cache systems where requests are allowed to fail, since the client can always fallback to the lower layer. This solution has two main drawbacks. For memcached specifically it could result in stale data being returned to the client when the node recovers as it did not contain the operations executed while it was down, because memcached servers are not aware of the other servers they do not synchronize. Other limitation not specific to memcached but that also applies due to the use of consistent hashing for determining the location of a key, is that a server can become overwhelmed because we assign a hot key [2] to a server already near full capacity, causing cascading failures. To avoid this the system presented in [25] takes advantage of the fact that a cache server can start with an empty state and maintains a pool of idle servers that are used when a failure is detected. This solves the problem of overwhelming servers due to re-assigning hot-keys as they were idle and still absorbs most of the load from the lower layer during the failure (not all since it still needs to be consulted to fill the cache).

A more typical approach to handle failures is through replication, supporting up to a predetermined number of faults. The work presented in [23] explores replication using protocols similar to the ones used to improve reliability in storage systems, namely RAID and erasure coding protocols. As expected by the fact that replication is not supported in the original system, this limits scalability and results in increased latency, due to the extra synchronization and computations required by erasure coding algorithms.

2.3.2 Discussion

It is clear that adding a caching layer can improve performance overall. A cache can be used in two ways.

- Content can be stored closer to the users, typically static content. This can be done by using a local cache as it is the case with web browsers or through a CDN. Both options improve user-perceived latency for read operations. The former can also reduce bandwidth usage while the latter distributes the load of the system serving the content across various servers and regions as opposed to a centralized system. Distributing the load offers more resilience as failures in one region, such as network partitions due to power outages or devices malfunctioning do not affect other regions.

- A caching layer can be introduced inside the application architecture between the frontend servers and the databases layer. Adding a cache does not mean the system's performance will always increase if the system only contains simple database queries that can complete fast then the gain will be negligible and not worth the

[2] a hot key is particular fatal due to having a high access rate when compared to others

increased system complexity. A cache is best used to store the results of complex computations, these can be complicated database queries or computations, possibly involving the participation of various distributed storage services.

Memcached [10] performs well for small scale systems, offering high throughput. Although some limitations come to light when expanding the system across various regions and a considerable amount of servers, these are successfully addressed in the work presented in [25] where modifications to Memcached are proposed and relevant deployment choices are suggested by the authors who used them on a globally deployed Memcached system.

When considering moving computations and data to the edge that cache model is not appropriate as it is not meant to support operations that modify the data and only read operations are addressed. It works well for applications such as Netflix or YouTube which use CDNs, where users mostly access media files, as well as for applications that must do complex computations which take a considerable amount of time and the result is valid for a considerable time, such as daily rankings. It does not work as well for *data intensive applications* [3] where latency is extremely important, for example when users interact with other users that are close and any significant latency is easily noticed, such as in mobile games as in Pokemon Go.

One thing cache has in common with our work is that resources are limited, many devices on edge have very limited resources and these might not even be static, so eventually, we will need to remove data to free up space. The LRU strategy used by Memcached might prove useful in our work.

2.4 Consistency on Distributed Systems

In distributed systems, write operations need to be propagated among replicas, so depending on how we decide to do it, there might be different versions of a value stored across the replicas. A strongly consistent system behaves as if data was stored in just a machine, showing only the most recent and valid states of the data. However, this is not always possible.

The CAP Theorem [7, 13] states that only 2 out of the 3 following guarantees can be provided simultaneously for a distributed system:

1. Consistency: Every read operation shows the most recent and valid state
2. Availability: Having the system always online, even during failures.
3. Partition Tolerance: Operations do not fail nor return incorrect values during network partitions.

[3] A data-intensive application can be defined as one that relies heavily on user input so it is constantly interacting with the server, the user inputs might trigger operations that modify data (which is not possible when using a cache).

Usually, partition tolerance is a must-have property because partitions are unavoidable in large distributed systems. This leaves us with a choice between the other two properties. Traditionally consistency has been chosen by most applications, however, due to the recent growth of web applications where user experience is extremely important and systems reaching sizes never before seen, being globally available with millions of users, availability has gained popularity and various systems have been built to take advantage of it, mainly No-SQL databases, like Dynamo [27], Cassandra [19], ChainReaction [3], and COPS [24].

2.4.1 Strong Consistency

Strong consistency offers guarantees that are very useful to certain types of applications where implementing them in a model with weaker guarantees would be extremely complicated. By choosing consistency over availability, we can ensure that all operations have the same order from the point of view of all users, observing only consistent states of the data. This is useful when having an always up to date and consistent state is critical to the application functionality, simplifying the implemented logic due to being easier to reason over the state evolution, as it only moves forward.

Operations have to be ordered globally so that all users see them in the same order, all replicas must respect this order to keep the state consistent. This is the basis for the **Sequential Consistency** Model. Operations in this model do not have to respect the real order in which they were issued. If we want the order to respect the real-time (i.e., wall-clock) at which the operations were issued then we need **Linearizability**. This is the strongest consistency model. Although it is not widely used due to the high level of synchronization required to implement it and the performance penalty that comes with it.

2.4.2 Weak Consistency

By choosing Availability, the application logic will be more complex because there might be a conflict between write operations, various versions of the state can coexist, and we do not want the user to see previous versions of the state so extra logic to handle these cases is required on the application side. We will now talk about three models that provide weak consistency.

Eventual Consistency: This model tries to maximize availability, providing only one guarantee, which is that all replicas will eventually converge if there are no more updates. Operations might be seen out of order, can be out of date as various versions of data can coexist before replicas converge, allowing users to see a state that is inconsistent. A protocol to resolve conflicting updates is required, it can either be a simple rule such as Last Write Wins or application-specific logic. The latter was the strategy used Dynamo, the first major system to implement eventual consistency.

Causal Consistency [20] : In this model the values returned by a replica respect the causal order between operations. In other words, one operation must appear to have been executed after all operations that preceded it (i.e., if a value is updated from two to four, then the following read of that value must be four). This provides a stronger consistency than eventual, without compromising on high availability.

Reasoning over the state evolution is simpler, but the implementation becomes more complicated when compared to eventual consistency. This is due to the fact that causal dependencies between operations must be tracked and the ones that are related must be executed in an order with their dependencies across all replicas (or at least their effects must become visible to users in such an order). Another advantage is that causally independent operations do not require any ordering, they can be executed concurrently which improves the system's performance.

Causal+ Consistency [24] : Causal consistency does not order concurrent operations, meaning that if there is a conflict then it is possible for replicas to return two different values, even after receiving both all the operations, this happens because the operations are not commutative and may have been executed in different orders at each replica. Clearly this is not a good user experience as it would forever return a different value depending replica processing the request. Casual+ consistency builds upon causal consistency and solves this problem by enforcing convergent conflict handling, that is, all conflicts are handled in the same manner, independently of the order they reach each replica. This can be achieved with a handler function that is associative and commutative. A simple method like last-write-wins is valid, however, more complex procedures defined at the application level can also be used as handlers. The result is that clients always see the system evolving forward, unlike in systems with eventual consistency where old results might be returned or in causal systems where replicas could diverge.

2.4.3 Tracking Causal Dependencies

Efficiently and accurately detecting causal dependencies between operations is crucial to ensure good performance of the system. It enables concurrency, as independent operations can be executed concurrently, and reduces synchronization costs. Creating a global order is a simple and correct of providing causality, however, this approach does not offer concurrency as it blindly marks all operations as dependent, due to the fact that all operations have to be executed in the same order across all nodes, requiring a high amount of synchronization, which has a significant impact on performance on a distributed system. A better approach would be to track which operations influence others. Only operations that are dependant need to be ordered while others can be executed in any order.

2.4.3.1 Causal history

One simple way to ensure causality is by keeping track of causal history, which contains the unique identifier of the operation and a representation of the previously executed operations on the node. When an operation is propagated across replicas, the causal history is appended to the operation, it is then merged on the receiving node and a new event is created. Now we can simply check if two operations are causally dependent by verifying if one is included in the causal history of the other. Although this works in theory, in a real system the causal history will grow indefinitely, making it difficult to create an efficient implementation. However, we can exploit the following property to develop an optimization:

- If an operation B is included in the causal history of A, then all operations contained in the causal history of B will also be included in the causal history of A.

Vector Clocks: By taking advantage of the previously mentioned property, we only need to store the most recent operations executed from each node. Consider the following causal history (letters represent nodes and numbers represent the logical time) a1, a2, b1, b2, b3, c1, c2, c3, it can now be represented through a vector clock [2,3,3], creating a representation of constant size instead of growing indefinitely. It contains the same information as causal history and all operations a direct mapping:

- When a new operation is issued, instead of adding it to the causal history, it is only needed to increment the counter on the vector position corresponding to the local node.
- When a new operation is received, calculating the union of both sets, remote and local, is equivalent to creating a new vector clock that takes the maximum value for each position, in other words, the most recent version seen for each node.
- A is causally dependent on B, if and only if B's id is contained in A's causal history, with vector clocks the equivalent condition is: if there is at least one position on A's vector clock where the value is equal or greater than B's vector clock corresponding position.

2.4.3.2 Metadata Propagation

Metadata is the information usually transmitted along with the operations (i.e., unique identifier, causal history, vector clocks, etc), the replicas then decide which operations can be executed concurrently and detect conflicts with this information. Its size and accuracy have a direct impact on performance. Compress it too much, and the false dependencies increase considerably, affecting data freshness (how old the data is when it becomes visible to the client). Do not compress it and the size will grow indefinitely, affecting the throughput. GentleRain [12] compresses the metadata into a single scalar,

sacrificing data freshness. Saturn tries to optimize data freshness when compared to GentleRain. However, metadata is still highly compressed to keep a constant size, hence it still results in a high number of false dependencies. Cure [2] uses a vector clock with an entry per data centre, therefore the metadata is compressed but not as much as the former, penalizing system's throughput. Ideally, we would like to have both data freshness and high throughput.

By controlling how metadata is propagated to each server, we can control when operations are executed locally and enforce an order that respects causality. This is the strategy used by Saturn [6] to provide causal consistency.

2.4.4 Consistency for Edge Computing

Strong consistency offers limited performance and due to the CAP theorem limitations can not be guaranteed alongside high availability, which is important for systems distributed across a considerable amount of servers, nor partition tolerance which is important for geographically distributed systems that have servers running on multiple data centres. Weak consistency models became very popular in the Cloud Computing paradigm for these reasons. Considering Edge Computing, where applications can be run on the Edge, and servers are no longer located close to each other. The limitations of using strong consistency become more evident. Availability becomes even more important now, because the number of servers increases and also because the environment on the Edge is more unstable, servers can experience more failures. Ensuring good performance becomes harder since the links connecting the servers are in the Wide Area Network (WAN). This means that stability and available bandwidth are lower while latency is higher between some edge servers when compared to cloud data centres, where providers can have private high-speed links directly connecting data centres. Having the traffic go through the WAN also increases the chance of a network partition, thus ensuring partition tolerance becomes even more important if we want to provide an always-online experience. Considering these limitations, it is clear that in order to achieve high performance in a highly distributed system, that either lives entirely on the Edge or can integrate with data centres, a weak consistency model is the more adequate choice. However, this does not mean we can not have any guarantees over the data, as shown is COPS[24], causal+ consistency can be implemented without having a considerable impact on the system's performance when compared to eventual consistency.

2.5 Automated Location-Aware Data Placement

In the Cloud Computing paradigm, applications are distributed across various servers in one or more data centres, allowing for more efficient scaling. This means that multiple instances of the application server are created and the data is also distributed across various servers. How we choose to distribute data will have a direct impact on performance

and application's costs.

When determining the data placement we can divide the problem in two levels. The first one is how to organize the data across the various nodes inside a data centre, while the second only needs to be addressed for geographically distributed systems composed of multiple data centres, which have become more common due to the growth of public Clouds, and consists in determining in which locations each data item should be stored.

- **Inside data centres:** At this level we have to start by choosing if each replica will contain a full copy of the data or the data will be distributed among various replicas. For small systems it might be feasible to store the whole data in a single replica and then just have secondary replicas, in this case there's no data placement problem to be addressed but this strategy is not adequate for large systems. This leads us to the remaining option where data placement becomes relevant, which is splitting the data across different replicas. In this case we can either use random placement, which work well enough when the goal is solely to achieve load balancing, or we can use data relations to make an informed decision, these relations can either be determined from usage patterns or provided by the application.

- **Geographically Distributed Systems:** Storing a full copy at each data centre would be the most simple data placement solution, ensuring all data is close to all users, but this results in a high cost for storage and synchronization. As users are located in various regions, it seems obvious that to reduce user-perceived latency data should be stored in the closest data centre, or in the K closest data centres where K is the replication factor to ensure fault tolerance. However, this approach does not address the fact most applications today include either communication (social networks) or collaboration (Google Docs) between the users, hence the same data is accessed by various users that can be in different geographic locations. It could result in excessive replicas of the data through various data centres, causing high synchronization costs, or in transactions involving various data centres, which results in added latency and bandwidth costs.

  Another challenge is data inter-dependencies, that is, the update of an object can trigger updates on other objects as well. These objects should also be stored together whenever possible, as to reduce synchronization costs, including inter-data centre traffic.

  It is clear that defining the best data placement is quite complicated due to the various factors that must be considered, such as:

  – Dynamic access patterns, which can happen when new features are introduced or simply because users are not fixed in one location. Some type of applications may also promote dynamic access patterns, like a page in a social network increasing popularity.

- The amount of replicas that are instantiated for each data item to find balance between performance and synchronization costs.
- Resources limitations, namely storage and bandwidth.
- Load balancing.
- Co-locality.

2.5.1 Offline Data Placement

We will now introduce various data placement schemes that offer static data placements, that is, they aren't capable of autonomously adapting to changes in order to provide an optimal data placement.

When distributing data across multiple servers the first option that comes to mind is spliting the data equally across the servers. The most common way to do this is by partitioning the data using a constant hashing function, where each server is responsible for a subset of the whole keyspace[4]. This scheme is used in various popular datastores, such as Cassandra[19], Dynamo[27], Redis.

Although distributing the data across various nodes inside a data centre increases performance by allowing horizontal scaling and load balancing, there is a ceiling to the possible performance gain after which adding more nodes will not improve the performance any further[16]. This is the main downside of this scheme, as possible performance gains from storing items frequently accessed together in a smaller number of nodes have been overlooked by these systems.

Volley [1] proposes a centralized system to determine ideal locations for data items by processing the application requests logs. The fact that all the logs have to be processed in a data centre is a serious limitation of this system, due to the high processing time and bandwidth cost of transferring all the logs. In order to support changes in patterns, the process has to periodically be re-executed.

A different approach is proposed by the authors of [31–33], where the data placement problem is modelled into a hypergraph partitioning problem, while addressing the previously mentioned main problem of partitioning the data using random placement. The hypergraph represents the data relations and the placement locations are extracted by partitioning the hypergraph. The work presented in [31] defines the basis for data placement based on hypergraph partitioning. Similarly to Volley, the proposed scheme is centralized, all logs have to be aggregated at a location. It stands out when compared to Volley because of its focus on maximizing co-locality, which improves the performance when read requests include multiple keys [16]. It also supports choosing a set of ideal data centres when we wish to constrain the number of replicas available per data item, since replicating the content to all data centres is not desirable due to its high cost. In [32] the authors argue that the overhead of the centralized design presented in their previous

[4]The set of possible values returned by the hashing function.

work [31] is significant under the increasing scale of datasets and propose a distributed scheme which offers a probabilistic solution. Requests information is maintained locally at each replica in what the authors define as Sketches, a specialized data structure that represent properties of a stream, such as request rates. The proposed method allows some operations to be executed when handling requests on a replica, distributing the load. A centralized controller is still required to coordinate, obtain and process samples from the replicas. However, the overhead is very small when compared to storing and processing all the logs due to the use of Sketches.

Data placement strategies based on hypergraphs [31–33] use partitioning heuristics in order to partition the graph, this is because hypergraph partitioning is an NP-Hard problem. The tool used by the authors to partition the hypergraphs suffers from scalability issues and efficiency, as mentioned in [4], which proposed a different partitioning scheme to achieve better performance. However, it does not support a dynamic number of replicas, a desirable feature to have when the usage patterns are dynamic and we want to adapt our deployment.

2.5.2 Online Data Placement

2.5.2.1 Data Placement in Data Centres

The work presented in [17] proposes a system to manage data and jobs on geographically distributed data centres, it tries to minimize costs by decreasing inter data centre traffic. Data tables are time partitioned, that is, each data partition corresponds to data generated during a period of time. The partitions can be independently replicated, this saves on inter data centre traffic and storage as only the relevant part of a table is replicated. Based on these partitioning scheme, the system supports configuring replicas to store only partitions of a table from a recent period of time, automatically deleting the partitions from the replica when they are no longer required. Jobs can also be moved to be executed at other data centres, this reduces inter data centre traffic by choosing the data centre that requires fewer data to be loaded from remote data centres.

In [35] an online scheme is proposed to minimise user latency by optimizing replica placement and replication on data centres for online social networks applications. It takes advantage of information provided by the application to identify client to data relations, which will determine the ideal locations for the data. It is capable of dynamically handling changes on the social graph, such as the removal of a friendship. The cost of storage and inter data centre communication is taken into consideration and the result is a placement that respects the latency bounds that were pre-defined while minimizing the total costs. In order to improve performance, co-locality is also optimized. It strives to maintain a balance between the number of replicas created to improve data locality and the cost of synchronizing such replicas that are placed in different data centres.

2.5.2.2 Data placement for Edge Computing

The already presented works have studied the data placement problem inside data centres and for geographically distributed data centres on the public Cloud, where the number of available data centres is fixed and not very large when compared to the number of available edge devices. They all share the same motivation, improve overall performance by moving data closer to where it is required, some also consider improving co-locality, both are still valid goals when optimizing the data placement on the Edge. However, these works focus on a global choice over where data is placed, which doesn't align very well with the properties of the Edge environment, which requires a more fine-grained control.

The number of devices on the Edge is considerably larger when compared to the Cloud, this combined with the fact that each Edge device will focus on serving the closest users means that we don't need to consider the global scope when determining which data to store on a device but require instead high flexibility as the set of nearby users isn't static as well as the limited available resources on the device (which means that eventually some data might have to be removed locally in order to free up space for more relevant data considering the currently active users).

2.6 Distributed Databases

With the global scale that web services have grown into in the past years, running a database on a single node was not sustainable due to the amount of data generated, so it was necessary to have a system that could distribute the data among various nodes while behaving as it was just one.

Typically databases focus was on ensuring ACID properties, however, efficiently implementing this in a distributed setting proved to be difficult due to the high amount of synchronization between the nodes required to ensure consistency. This led to the appearance of the Key-Value Store model, which dropped the consistency from the ACID properties, offering no guarantees about the consistency of the state, moving that responsibility to the application side. Although the application's complexity could increase, the main advantage was that the overhead introduced by the consistency guarantee was only present when required, enabling better scalability overall. The set of supported operations was much more restrictive when compared to the SQL interface. Initially only get/put operations over a single key and primary types were supported, like in Dynamo. However, most recent systems have started offering more advanced features. Such as structured data seen in Cassandra, and read transactions seen ChainReaction and COPS.

Fault tolerance is also important, hardware faults are inevitable events when the system scales to a considerable number of nodes. Therefore data needs to be stored simultaneously in various nodes, also called replicas. When the number of replicas is considerably small, simpler strategies like primary-secondary backup are good enough.

But these do not scale well for very large systems, a Multi-Master strategy (introduced in Section 2.2.2) provides better scalability, although considerably more complex to implement.

There are various approaches to build a distributed database system that can scale to a global level by using one of the weak consistency (introduced in Section 2.4.2) models to maximize availability. We will now introduce various distributed database systems that strive to offer high availability, discuss their strategies and corresponding pros and cons.

2.6.1 Dynamo

Dynamo targets applications that do not require most of the features provided in typical SQL databases, such as complex queries, transactions and the consistency provided by the relational model. If offers a simple key/value interface, supporting only get/put operations over a unique key (operations over multiple keys is not supported). It primarily focuses on high availability.[27]

Consistency and high availability: In order to ensure the maximum high availability possible, it provides eventual consistency. This allows the system to expose an always online behaviour, even though failures can be occurring in the background. Operations are never rejected, the system is "always writeable". This feature is critical for various applications, where if the write failed it would lead to poor user experience. For example, the Amazon shopping cart, if adding items to the cart failed and the user had to add it again, this would not be a very good experience. The solution is to push the conflict resolution complexity to be executed during the read operations.

To ensure consistency, a protocol similar to those used in quorum systems is used. Two numbers must be defined, R and W, indicating the minimum number of nodes that must participate in read and write operations respectively. When R+W is greater than the number of nodes in the system, the behaviour is equivalent to a quorum system.

Conflict Handling / Versioning: Dynamo exposes two mechanisms to handle conflicts. It can be done at the database level, but since the system has no knowledge of the data structure it can only provide a basic method, which is Last Write Wins. The alternative is to do it at the application level, which allows much more flexibility since the application can make a more informed decision over what is the best solution.

Version vectors are used to track the evolution of object versions and detect concurrent updates. Read operations contain the corresponding version vector, which must be stored by the client and sent with write operations.

When Last Write Wins policy is being used, conflict resolution is trivial. In the other case, when application level is being used, all conflicting versions are stored on the system temporarily. Write operations can be executed during conflicts, the context provided

contains the vector clock which allows Dynamo to determine which version the client is updating. On the next read operation, all the versions are sent to the application, It must then execute the desired logic over the returned versions to resolve the conflict and return a new version that will replace the others as the correct value. The old conflicting versions are deleted when the new version is received.

Partitioning: The data is partitioned using a variant of consistent hashing, with virtual nodes where each one is responsible for a range of keys of equal size. Each real node is responsible for a number of virtual nodes. This has various advantages:

- Each real node can vary the number of virtual nodes it responsible for, depending on its computing power and resources. Nodes might have different hardware configurations.
- When a node stops responding, its load is distributed evenly across the remaining nodes, as opposed to being completely handed to the closest node.
- When a node joins the system, it is assigned multiple virtual nodes for which he is responsible, providing uniform load and data distribution.

This works well for Dynamo since it does not support operations over multiple keys. However, in an application where there is data that is frequently accessed together we might want to provide an operation to read multiple keys at once. With this partitioning scheme, related data items would probably be spread among various nodes, affecting performance. Decreasing the number of nodes participating in a request typically results in a faster response time.

Membership: A Ring Membership is used, each node is assigned a unique id that is within the range of the consistent hashing function used. Nodes can only be added and removed explicitly by an admin connected to a Dynamo node. Updates to the membership are propagated with a gossip-based protocol that ensures an eventually consistent view. Partitioning and placement information is also propagated this way, allowing each node to communicate directly with any node in the system.

Replication: Each data item is replicated at a pre-configured number of replicas, referred to as N. Based on the key, a coordinator node is determined. This node is then responsible for ensuring the replication of the data item to the successor N-1 nodes (a ring topology is used since partitioning is based on consistent hashing). Any node in the system can determine a preference list containing the nodes where a key is stored.

Due to the use of "virtual nodes", there is a chance that in the first N successors for a key there are less than N physical nodes. The method to build the preference list skips these nodes to ensure that it contains only distinct physical nodes.

Failure Detection: When a node fails to respond to a message sent by another node, it is locally marked as unresponsive. Alternate nodes are used while periodic attempts to re-establish a connection are done in the background. This is enough to avoid attempts to communicate with failed nodes. In case two nodes never exchange messages, they will never be aware whether the other is failing or not. When a node fails to respond to a client request, an alternate node is contacted and a hint is included in the metadata indicating the unresponsive node. Upon recovery, the alternate node will transfer the operations he received during the failure. The alternate node will check periodically when the originally intended node recovers, and then proceed to transfer the operations he received meanwhile.

2.6.2 Cassandra

Cassandra is a key-value store that is based on Dynamo. It provides eventual consistency and high availability, being capable of scaling to hundreds of nodes. Although it does not provide a relational model, it offers a structured data model with support for tables with complex data types. Which is where it stands out when compared to Dynamo.

Data model: Data can be organized in tables, which are represented by distributed multidimensional map indexed by a key. Tables are composed of columns, which can either be simple columns or Super Columns. Super columns are simply a set of columns, it can be visualised as a Dictionary that has Dictionaries as values. This allows for a complex data structure which can maintain tables, and these tables can themselves have properties with complex data types like Dictionaries, Sets, and Lists.

Replication: Cassandra replicates data based on a replication factor that determines the total number of replicas per data row. It offers the following two replication strategies:

- **Simple Strategy:** Similarly to Dynamo, each key as a coordinator node and it is simply replicated to the N-1 successor replicas, where N is the replication factor. This strategy is meant for simple deployments using only one data centre.

- **Network Topology Strategy:** This strategy is adequate for larger deployments across various data centres. It has access to the deployment configuration and attempts to place replicas on different racks, providing better fault tolerance. Also allowing the configuration of the number of desired replicas per data centre.

Partitioning: Each node is assigned a value from the output of a consistent hash function. The node is responsible for all the keys between itself and its predecessor. One particularity of Cassandra is that the hash function used is order-preserving. This allows columns to be sorted by time or by name, in whichever order the applications specify.

Membership: Similarly to Dynamo, nodes are organized in a Ring and each one has a unique id within the range of the hash function used. Membership information on Cassandra is propagated using a gossip-based mechanism. It is based on Scuttle-butt[28], due to its efficiency on CPU utilization and usage of the gossip channel.

Nodes can be added through any node, by an admin connected using a web interface or a command-line utility, just like in Dynamo.

Failure Detection: Cassandra makes use of a gossip process to track other nodes state. Based on the state it each node locally determines a value which will then be compared against the threshold determined by a modified version of the Accrual Failure Detector[15] to calculate a per-node threshold that considers network performance, workload and historical conditions (i.e., how often it failed recently). If the value is greater then the node is marked as being down. When compared to typical detectors that output a boolean value, Accrual Failure Detectors have the advantage of adapting well to network and server load conditions as the threshold is dynamic and calculated per-node, while still offering good accuracy and speed.

2.6.3 COPS

COPS[24] is the first system built with casual+ consistency in mind, the strongest possible model without compromising on partition tolerance, ensuring high availability. Its primary focus is on providing causal+ consistency without sacrificing the scalability typically only seen on a system with eventual consistency. It also introduced a new operation get transaction which can return a consistent view of multiple keys distributed across various nodes in a data centre, which separates it from the previously discussed systems. The system is designed to support a small number of large scale data centres, with one cluster per data centre. Each cluster then contains frontend servers (clients of COPS) and back-end servers with key-value stores.

Causal+ Consistency: COPS depends on storing metadata for each key pair value, which contains the value, version and list of dependencies. A garbage collector is used to remove old metadata that is no longer required to stop the size growing indefinitely.

It also depends on clients maintaining a context with information to determine the current causal dependencies of operations which must be sent with all requests. Operations are only applied after all their dependencies have been committed.

Scalability: Similarly to previous systems, consistent hashing is used to partition the data. Each cluster is partitioned independently, each key has a primary node per cluster and each node is responsible for a set of key ranges. Operations are linearizable across nodes in the local cluster and return as soon as they are completed locally. Performance wise, using linearizability in a cluster is acceptable due to the low latency observed within

data centres and the low chance of a partition occurring inside a data centre (modern data centres implement redundant routing). Communication between clusters is done in an asynchronous way.

Replication/Fault Tolerance: Chain-replication is used as described in Section 2.2.2. Write operations are marked as committed when they reach the tail. After a write is completed in the local cluster, it is asynchronously sent to the corresponding heads in other clusters, along with its dependencies. Before each operation is executed, all dependencies are checked if they have been committed in the local cluster. Only then is the operation applied and becomes visible. Read operations are sent to the tail and are executed locally. Data centre failures are supported but have some limitations. First, all the write operations that were not replicated are lost if the failure is permanent, and second, replication queues and metadata information in active data centres will grow indefinitely until the failure is over or the data centre is removed from the membership.

2.6.4 ChainReaction

ChainReaction[3] is a geographically distributed key-value datastore, which uses a modified version of chain-replication to provide causal+ consistency using a minimal amount of metadata, which is maintained on the client-side. It also provides read transactions to obtain a causal+ consistent view of multiple keys, similarly to COPS.

Partitioning: Consistent hashing is again used to partition the system and a ring topology is used to assign data items to servers. As a variant of chain-replication is being used, some behaviour diverges from typical consistent hashing systems. First, during replication instead of replicating the values to the N-1 successors in any order, these nodes form the chain to be used with the chain-replication scheme. Secondly, read operations are directed to the tail of the chain, unlike in other systems, all operations related to a key would be sent to the coordinator. A one-hop Distributed Hash Table (DHT) is maintained at all nodes to ensure that nodes can locally map keys to nodes, without having to use DHT routing.

Chain-replication variant: The replication scheme used by this system is a variant of chain-replication, introduced in Section 2.2.2. The original version is limited by the fact that it provides linearizability, it uses the tail node as a serialization point, hence it does not leverage replicas in between to participate in load balancing. The variant used instead offers causal+ consistency which allows better use of all replicas in the middle of the chain. It allows other replicas besides the tail to respond to read requests and can be used similarly to a quorum system, where operations only return after being replicated to the first K replicas. This variant allows adding more replicas to ensure better durability and load balancing without impacting the latency of write operations, as values are lazily propagated down the chain (except for the K first replicas, to ensure low latency).

Causal+ consistency: In order to ensure causal+ consistency, write operations are delayed until all the dependencies have propagated to all the chains locally. This guarantees that the user will not see old versions of the dependencies, as required by causal+ consistency. Dependencies are included in the metadata provided by the client in the operation request.

Geo-Replication: Communication between data centres is done asynchronously. Operations are ordered globally and concurrent operations can be executed on various data centres. When concurrent updates are conflicting the basic last write wins rule is used to solve them. For this purpose, every operation has a timestamp and the data centre identifier which can be used to order the conflicts.

Values are replicated to other data centres to ensure fault tolerance, but in case of a data centre failure the updates that were not propagated can be lost if it never recovers. The system can be adapted to only return to the client after the value is confirmed as replicated to other data centres instead of just waiting for replicas in the local data centre. Another approach would be to have the client resubmit the operation request to another data centre. This can add a significant amount of latency but can be useful with critical update operations as an alternative.

Get Transactions: Introduced in COPS, ChainReaction borrows some techniques and improves on others. It uses multi-versioning, as does COPS, to be able to read versions consistent to causal+ while concurrent writes are happening. This allows the operator to be non-blocking. This operation can be done in one round except when dependencies include operations from other data centres that have not yet reached the local data centre, where a more complex two-phase protocol is required in order to complete the request.

When compared to COPS, the scheme used for transactions in ChainReaction fares better [3]. It requires only one round most of the time, unlike COPS where two rounds are required every time values are updated during the transaction.

2.6.5 Redis

Redis is an in-memory data store with a focus on performance, it is completely written in C. It offers native support for complex data types, such as list and sets, as well as basic data types.

Redis can be configured to work as a database, but is mostly used as a cache. It offers support for automatic key eviction using either a LRU or Least Frequently Used (LFU) strategy that will kick in when the used memory reaches the configurable maximum value.

Replication: A primary-secondary strategy is used, where the secondary replica is configured to be an exact replica of the primary. It receives a stream of commands from

the primary, upon a link failure the connection is re established and if required a partial synchronization process is executed in order to received missed commands. One primary node can have multiple secondary nodes if desired.

The replication is done in an asynchronous way by default, however Redis offers the WAIT command that will wait for a desired number of replicas to have acknowledged all the previous writes for a certain amount of time, or forever if the timeout specified is 0. Although this considerably reduces the chance of data loss in case of a failure, it does not ensure strong consistency across the replicas and it is still possible to lose a synchronous write.

The system scalability can be improved by allowing secondary replicas to handle read operations.

Partitioning: Redis has a Cluster mode which allows splitting the data across various nodes. The keyspace is split into 16384 hash slots, where each primary node handles a subset of the slots. This effectively limits the cluster size since each primary node must handle at least 1 slot, however each one can have multiple secondary replicas that can be configured to take over in case of a failure.

The cluster can be reconfigured on the fly, which will cause the keys to be moved from one node to another. When the system is stable each hash slot will be served by a single node.

Redis does not allow the application to choose the node where data is stored, however it provides a mechanism to ensure that keys are stored together in the same node by limiting the part of the key that is used to determine the hash slot. This is useful for cases where we know that certain data is correlated and will frequently be requested together, and specially because transactions involving multiple nodes are not allowed, hence all the keys involved in a transaction must be stored in a single node.

Fault tolerance: Redis has an internal system, Redis Sentinel, that can be configured to detect and react automatically to failures in system. Redis Sentinel is a distributed system that monitors the master nodes in the Redis deployment. The Sentinel system must be composed by at least 3 instances in order to work properly, that is, it must be possible to have a majority of nodes that agree of a fact, such as if a master node is currently down. When a failure is detected by a configurable number of sentinel instances, one of the instances will try to start a failover to promote a replica to master, this always has to be allowed by a majority of the sentinel instances. The clients have to contact the sentinel system in order to discover the current master nodes.

Disk persistence: By default Redis does not persist the data to the disk, however depending on the application this might be a desired behaviour. For this reason Redis provides two different methods to persist data. The first one is by using an Append-Only File (AOF) (i.e., a sequential log), it consists in writing executed commands that changed

the database state into a file, which is then used to recreate the exact database state. The second method, named RDB (Redis Database) , consists in creating point-in-time snapshots of the database into a binary file, that is, a representation of the in-memory state content is stored on the file as opposed the previous method that stored the command that generated the state. How often the snapshot is created is configurable by providing a time period and the minimum number of operations to trigger the snapshot creation, it is possible to indicate more than combination to trigger the snapshot creation.

RDB (Redis Database) : This method works great to be used as a backup meant to be archived, it can be used as a restore point in case of disasters, such as restoring the database to an old state and wrongly deleting relevant data, and provides faster restart times since only the data needs to be loaded into memory as opposed to the AOF method where all the commands have to be executed in order to restore the state.

AOF (Append Only File) : If the focus is on minimizing data loss in case of an unexpected failure then the AOF is more adequate as it will always be more complete due to the fact that every command is logged to the file as opposed the periodic write seen on the RDB method.

2.7 Summary

In this chapter we presented all relevant related work that was studied and used as a basis for this thesis.

In the next chapter we will present the proposed algorithms to address the challenge of replication data in a dynamic environment and detail the reasoning behind them.

Chapter 3

Causal consistency with dynamic data placement

In this chapter we present and explain the various algorithm that are part of our solution for partial replication that addresses the following challenges:

- Dynamic creation and removal of data replicas.
- Autonomous removal of unused data replicas.
- Ensure causal consistency

The presented algorithms are designed in a way that focus only on the main purpose of the algorithm. We are interested in a solution that allows flexibility in the locations where and when replicas are created, so that replicas can be managed dynamically.

We start by providing and overview of our solution and why it works in section 3.1. We then define the system model and the assumptions upon which we build our work in Section 3.2, in which we present the main components that are part of our solution. In Section 3.3 we introduce how the overlay containing the nodes in the system will be defined and how the nodes will communicate. In Section 3.4 we define and explain the algorithms used for managing replicas dynamically. Followed by Section 3.5 where we define the algorithms required to ensure causality in our system. In Section 3.6 we explain the process that allows the overlay to adapt when a failure is detected. Finally in Section 3.7 we explain how and why the client is an important part of the system in order to ensure that the causality guarantee is not broken.

3.1 Overview

In our solution we define two different types of replicas, the stable replicas which are located in the cloud and the dynamic replicas which are located in the Edge, the communication links between the replicas form an overlay that is an oriented acyclic graph, similar to a tree structure. The dynamic replicas contain only a subset of the whole keys in the system, however it is relevant to note that the keys present in a node are always present in all the nodes above him, that is, all the nodes in the path between and the cloud. This is due to the fact that messages are only propagated using the links in the overlay, and the the fact that read operations for existing keys and write operations cause a new replica for the key to be added in all the nodes it executes in, this can include all the nodes in the path between the current node up to the cloud.

Each node maintains a version vector for the nearby nodes, keeping the size small relatively small, this is possible because when an operation is propagated it increments the version in all the nodes where the operation is replicated to, this way it is possible to detect concurrent operations without having to keep track of all the nodes in the system, which is something that would not scale with a large number of nodes in the system. Upon detecting concurrent operations, a deterministic conflict resolution is applied, this together with the use of First In First Out (FIFO) links and only sending messages through the links defined in the overlay ensures causal+ consistency, where each client never sees a previous version of the system than the most recent one he saw so far.

When data is unused for a certain amount of time it starts being remove from the dynamic replicas, starting at the nodes at the bottom of the overlay, until that data is stored only in stable replicas, where all the system data is stored, being only removed when executing delete operations.

3.2 System Model

First, a relevant aspect for our solution is that we assume that the connections established between the nodes behave as a FIFO queue, this is extremely important so that messages are never delivered in a different order from the one in which they were sent.

The failure model assumed is fail-stop, so when a node fails it stops sending messages until it is restarted. When a node is restarted it behaves as a new node in the system. All the nodes in the system can be trusted and byzantine faults are not considered.

The system is composed by 3 main components, the redis replicas, a client library that must execute some logic on the client side and a discovery service that has relevant metadata for all the nodes in the system, used by the clients to obtain information, namely which node should they connect to based on their location.

Redis Replicas: There are two different types of replicas, stable or dynamic , they offer the same functionalities but differ on how they function, namely a dynamic replica must

always have one seed node, in case the seed node is another dynamic replica, or one or more seed nodes in case it is a stable replica and the replicas in the data centre are configured in cluster mode. When configured in cluster mode the keyspace is split across the multiple redis instances during the configuration process and it does not change later. In more detail :

- **Stable Replicas:** These replicas are defined by having a stable key space for which they are responsible, that is, each replica will always store only data for a set of keys defined during configuration(i.e., before they are started). They are located inside data centres and have a considerable amount of resources available, which doesn't change over time, we further assume that this replicas are available most of the time The total system data is distributed across these replicas, meaning that any data stored in the system can be accessed through one of replicas of this type.
- **Dynamic Replicas:** Opposite to stable replicas, the key space for which this replicas are responsible can change dynamically over time, it will always be a subset of the data present in its seed node, they adapt according to the usage patterns of users and the available resources(in the edge node where they execute). These replicas run on the edge, which is composed of various types of devices that can have very different characteristics.

Client: The client is its own component because it is not a stateless client, it will need to analyse and keep track of metadata received with each response, this metadata will be required in order to connect to a different node without breaking the causality guarantee. When redis is configured in cluster mode, the client is responsible for sending the commands to the correct nodes(i.e, it is aware of the sharding of data within a data centre).

Discovery Service: This component contains information about system membership, and contains relevant information about each node, such as their network location and GPS coordinates of their location. Its responsibilities are resolving the replicas identifiers(ids) into network locations and allowing clients to query what is the more adequate node for them to connect to, based on their location.

3.3 Overlay Structure

The nodes are organized in a oriented acyclic graph, similar to a tree, however at the root level there may be more than one node, this is because the root represents a data centre, where redis can either be running as a single instance or as in cluster mode with multiple instances, where each instance is represented by a node. Another particularity of this overlay is that the child nodes must have exactly one incoming link, except the ones that are directly under the root, in which case they must have exactly one incoming link from

each node at the root. An example can be seen on Figure 3.1. We note that we do not explicitly address scenarios with multiple data centres, but such extension is easy and is beyond the scope of this work.

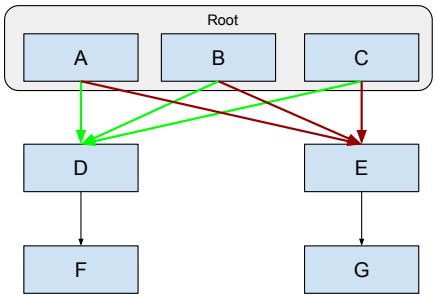


Figure 3.1: Overlay

The nodes only need to have a partial view of the system, based on a configurable radius(defined in number of hops in the logical network between nodes), this reduces the amount of metadata that each node needs to maintain and propagate, either between them and to the clients. This partial view can be defined as the set of nodes that can reach the current node or are reachable from the current node by a path with a distance less than or equal to the defined radius. An example of the view of each node in the overlay from the Figure 3.1 is presented in Table 3.1, considering a radius of 2.

Node	View
A	D E F G
B	D E F G
C	D E F G
D	A B C F
E	B C D G
F	A B C D
G	A B C E

Table 3.1: View per node for the overlay in Figure 3.1, with radius defined as 2

The nodes communicate using only the links present in the overlay, this and the fact that the communication channels preserve the order in which messages are sent makes it easier to guarantee causality, because the order between two messages sent by a node, let's say message A and B, where A is sent first and is then followed message B, will always be preserved when they are propagated between other nodes, it is not possible for B to be received before A, which is required to ensure causality.

Algorithm 3.1 Dynamic Data Placement

State:
- subnodes
- localKeys ▷ set containing all the locals keys
- subscriptions ▷ sub nodes subscribed per key
- expires ▷ set containing the keys with an expiration time
- seedNode

Interfaces:
- readFromSeedNode(key)
- getKeyValue(key)
- saveKey(key,value) ▷ Save a new key to the local db

```
1:  procedure READ(sender, key)
2:      if key ∈ localKeys then
3:          if sender ∈ subnodes then
4:              subscriptions[key] ← subscriptions[key] ⋃ [sender]
5:              if key ∈ expires then
6:                  expires ← expires \ key
7:          return getKeyValue(key)
8:      else if seedNode ≠ ∅ then                ▷ Check if it is an edge node
9:          reply ← readFromSeedNode(key)
10:         if reply ≠ ∅ then
11:             saveKey(key, value)
12:             localKeys ← localKeys ⋃ [key]
13:             if sender ∈ subnodes then
14:                 subscriptions[key] ← subscriptions[key] ⋃ [sender]
15:         return reply
16: procedure WRITE(sender, key, value)
17:     if key ∉ localKeys then
18:         localKeys ← localKeys ⋃ key
19:         if sender ∈ subnodes then
20:             subscriptions[key] ← subscriptions[key] ⋃ [sender]
21:     saveKey(key, value)
22: procedure RECEIVEUNSUBSCRIBEMESSAGE(sender, key)
23:     if key ∈ localKeys then
24:         subscriptions[key] ← subscriptions[key] \ sender
25:         if subscriptions[key] = ∅ then
26:             expires ← expires ⋃ [key]
```

3.4 Dynamic data placement

As mentioned before, we have two different types of replicas, stable and dynamic, at the start the only replicas that contain any data are the stable replicas. Dynamic replicas are initialized without any keys stored locally and a configured seed node, then as commands are processed locally the replica figures out which keys it should maintain locally and request their values from the seed node if required(as shown in the READ and WRITE procedures defined in Algorithm 3.1). These keys are cached locally and added to the *local keys* set, meaning operations over these keys can be done without consulting the seed node again. by the replica some keys will start to be cached and marked as local keys.

The *local keys* at each replica can change due operations done by clients, more keys can be added due to read operations for an existing key that is not present locally or write operations for a key that was not stored locally, we go into more detail in Section 3.4.1. It can also decrease when executing a delete operation for an existing key or as an optimization due to the lack of operations executed over the key as well as the lack of available resources, which we will go into detail on the Section 3.4.2

3.4.1 Creating new key replicas on edge nodes

- **Read:** As shown in the *READ* procedure defined in the Algorithm 3.1, when receiving a read request, we must check if it is for a local key, if yes then the value is cached locally and the response can be sent immediately, however if it is not then there are two possible scenarios. Either the current replica is a stable replica, which means the key does not exist, or the current replica is a dynamic replica, in which case it must send a read request to the seed node and upon receiving the response use it as the result to complete the operation. If the response contains the value for the key it is because it exists, so the value will be cached locally and the key added to the local keys, and if the command came from a sub node(i.e., a child node in the overlay defined, either directly or indirectly linked) then that same node must be added to the key subscriptions. Only after executing this checks will the response be sent to the client.

- **Write:** As shown in the *WRITE* procedure defined in the Algorithm 3.1 , When the key is not already a local key, it is added to the local keys on the current node immediately, allowing sub nodes to subscribe to this key, the key value is cached locally in order to allow read requests to be executed without the need to send a request to the seed node. This is done independently of the command origin, whether it originated directly from a client or if it is being executed due to an operation propagated by a sub node. The latter has an extra step which is adding the sub node to the list of subscriptions for new key.

3.4.2 Removing replicas on edge nodes

There are three scenarios where a key needs to be removed from an edge node. The first is simply when a key is deleted, either by a command issued by client or from a command propagated from another node. The second is when the key expires after being unused for a period of time (hence reducing the amount of existing replicas for a key, which reduces the amount of messages generated when its value is modified) and the third is when the server is running out of memory. Independently of the reason the key is being removed from the local keys, the process for removing the key always contains the following:

1. Delete the cached value for the local key.
2. Remove the key from the set of local keys.
3. Notify the seed node to remove the subscription for the key.

If the cause for the key removal is a delete command, then it is also necessary to propagate the command to the sub nodes subscribed to this key and to the seed node, with the exception of the node that propagated the command, if it came from another node and not directly from a client.

3.4.2.1 Suppressing replicas on unused data

As we keep adding more local keys, the memory occupied needs to be controlled as the resources on dynamic replicas are limited. In order to keep the memory occupied under control, some keys may have to be removed, however a key can only be removed if there are no pending operations and no active subscriptions from sub nodes for that same key, this is required to minimize the risk that operations executed over the key are not lost.

We can have active and passive key removal, the former consists of removing the key-pair locally after it's unused for a certain configurable period, the later is triggered when the occupied memory surpasses the configurable limit and will start removing local keys until enough memory is freed.

Active key removal is useful in order to reduce the messages exchanged across the system, hence reducing the overhead by not propagating updates for keys to nodes where it's not very likely that the key value will be read.

3.5 Causality

When a key is stored on multiple nodes it is possible that a node executes an operation before receiving the update containing the most recent value for a key, in this case we say that the operations are concurrent. In order to detect these operations we created a solution based on version vectors [20], which is described in the Algorithm 3.2, where we define how to detect and handle concurrent operations, and Algorithm 3.3 where we define how the metadata is propagated across the nodes.

3.5.1 Adapted Version Vectors

The typical behaviour of a version vector when executing an operation is to increment the version corresponding to the current node and update the corresponding entry in the version vector of other nodes when propagating the operation. The most relevant thing to note is that an operation only causes the entry of one node to be incremented and then updated across all the version vectors. This however, requires the version vector to have one entry per replica, which has a non-negligible overhead for a large number of replicas.

We propose to alter this behaviour to also increment the version of the entry of a node when receiving the propagated operation. Due to the way we propagate operations, using only the links established in the overlay, meaning that a propagated operation will only be received through one link, we can preserve the information that an operation was executed and detect concurrent operations, without requiring that each node keeps an entry in its version vector for all the replicas in the system. To ensure that concurrent operations can still be detected it is only required to keep entries in the version vector for directly connected nodes, both below and above the tree topology. Because any operation executed in the system will increment the version in the entry for at least one of the nearby nodes, reasoning about concurrent operations when analysing the version vector is done in the same way as if there was an entry for all the nodes in the system. The only information that is lost in this process is the node where the operation was initially executed, instead we can only know which one of the nearby nodes received the propagated operation, this is not problem as it is still possible to detect concurrent operations using this information, in exactly the same way as before.

Hence, when a node with id **A**, with the version vector **[A:2; B:3]** and whose only nearby node is **B**, receives an operation with a version vector from node **B** with values **[A:2; B:4; C:3]** it will update its own version vector after applying the operation so that it has the following values **[A:3; B:4]**, where we can see the version of node **A** was incremented while processing the received operation and the value for node **B** was updated to the corresponding value present in the received version vector. About the entry for the node **C**, it was discarded because it is outside the visibility radius defined for node **A**, hence it does not need to keep track of its version.

3.5.2 Detecting concurrent operations

As defined in the ExecuteReceivedCommand in Algorithm 3.2, when a node executes an operation from a client the version on the entry corresponding to that node is incremented in the local version vector, followed by adding the executed command along with some metadata to a list of unconfirmed commands. The metadata includes the resulting version vector after applying the operation and the hash of the key included in the command. This list is used in cases where a concurrent operation was detected in order to know if the operations conflict or not. A conflict is detected when the new command operates over the same key as one of the concurrent operations.

Algorithm 3.2 Processing Propagated Commands

State:

localVersionVector ▷ local node version vector
localNodeId ▷ local node id
commands ▷ list of volatile commands and metadata
subscriptions ▷ sub nodes subscribed per key
seedNode ▷ the seed node for edge nodes
subnodes

Interfaces:

getCommandKey(command)
compareEntry(versionVectorLeft, versionVectorRight, nodeId)
vvMerge(versionVector) ▷ Defined in the Algorithm 3.4
applyCommand(command)
updateVersionTracker(versionVector, nodeId) ▷ Maintains the version of this node that all nearby seed nodes have acknowledged

updateGlobalVersionTracker(versionVector, nodeId) ▷ Maintains the version of this node that all the nearby nodes have acknowledged

```
1:  procedure ReceivePropagatedCommand(sender, command, versionVector)
2:      propagateToSender ← false
3:      senderId ← sender.nodeId
4:      updateGlobalVersionTracker(versionVector, senderId)
5:      if sender = seedNode then
6:          updateVersionTracker(versionVector, senderId)
7:      cmpLocal ← compareEntry(localVersionVector, versionVector, localNodeId)
8:      cmpSender ← compareEntry(localVersionVector, versionVector, senderId)
9:      if cmpLocal > 0 then
10:         if cmpSender < 0 then          ▷ If true then there are concurrent operations
11:             propagateToSender ← checkConflicts(key) & senderId ∈ subnodes
12:             ExecuteReceivedCommand(sender, command)
13:             PropagateCommand(command, sender, propagateToSender)
14:     else if cmpLocal == 0 & cmpSender < 0 then
15:         ExecuteReceivedCommand(sender, command)
16:         PropagateCommand(command, sender, propagateToSender)
17:
18: procedure ExecuteReceivedCommand(senderId, senderVersionVector, command)
19:     versionVector ← vvMerge(senderVersionVector)
20:     newKeyCreated ← applyCommand(command)
21:     localVersionVector[localNodeId]+ = 1
22:     commands ← commands ⋃ (command, localVersionVector)
23:     if newKeyCreated then
24:         key ← getCommandKey(command)
25:         subscriptions[key] ← subscriptions[key] ⋃ [sender]
26:
27: procedure PropagateCommand(command, sender, propagateToSender)
28:     key ← getCommandKey(command)
29:     for all node ∈ subcriptions[key] do
30:         if node ≠ sender || propagateToSender then
31:             propagate(node, command)
32:     if sender ≠ seedNode then
33:         propagate(sender, command)
```

Algorithm 3.3 Processing Metadata Updates

State:
localVersionVector ▷ local node version vector
localNodeId ▷ local node id
commands ▷ list of volatile commands and metadata
seedNode ▷ the seed node for edge nodes
subnodes

Interfaces:
vvMerge(versionVector) ▷ Defined in the algorithm 3.4
updateVersionTracker(versionVector, nodeId) ▷ Maintains the version of this node that all nearby seed nodes have acknowledged

updateGlobalVersionTracker(versionVector, nodeId) ▷ Maintains the version of this node that all the nearby nodes have acknowledged

```
1:  procedure ReceiveUpdateMessage(sender, originalSender, versionVector)
2:      if originalSender = sender then
3:          vvMerge(versionVector)
4:      else
5:          localVersionVector[originalSender] ← versionVector[originalSender]
6:      updateGlobalVersionTracker(versionVector, originalSender.id)
7:      if sender = seedNode then
8:          updateVersionTracker(versionVector, originalSender.id)
9:          for all node ∈ subnodes do
10:             filteredVersionVector ← filteredVersionVector(localVersionVector,
11:                                                           localNodeId, node)
12:             send(UPDATE, node, filteredVersionVector)
13:     else if seedNode ≠ ∅ then                    ▷ check if it is an edge node
14:         send(UPDATE, seedNode, localVersionVector)
15:
16: procedure FilterVersionVector(versionVector, localNodeId, destId)
17:     filteredVersionVector ← ∅
18:     for all entry ∈ versionVector do
19:         if entry.id is a seed node then    ▷ Include all the visible nodes above this node
20:             filteredVersionVector ← filteredVersionVector ⋃ [entry]
21:         else if entry.id = localNodeId then
22:             filteredVersionVector ← filteredVersionVector ⋃ [entry]
23:         else if entry.id = destId then
24:             filteredVersionVector ← filteredVersionVector ⋃ [entry]
25:         else if entry.id is sub node of destId then
26:             filteredVersionVector ← filteredVersionVector ⋃ [entry]
27:     return filteredVersionVector
```

When a operation is propagated the current version vector is sent as well, this allows the receiver node to know if any operation has been executed locally that the sender was not aware when executing the operation by comparing the local version vector with the version vector as defined in the *ReceivePropagatedCommand* procedure in the Algorithm 3.2. A propagated operation is determined to be concurrent by comparing the entry in both version vectors for the current node and the sender node(lines 7-8), if the entry for the current node contains a greater version in the version vector of the current node(line 9) but the entry for the sender node contains a greater version in the sender's version vector(line 10) then it is a concurrent operation that was propagated.

When concurrent operations are detected we have to determine which one is the final result. In our solution we decided to use the last write wins policy.

Algorithm 3.4 Version Vector Operations

```
    State:
        max_distance
        localVersionVector
        localNodeId
1:  procedure vvMerge(senderVersionVector)
2:      for all entry ∈ senderVersionVector do
3:          if entry.distance < max_distance then
4:              if entry.nodeId ≠ localNodeId then
5:                  if entry.nodeId ∈ localVersionVector then
6:                      localVersionVector[entry.nodeId] ← entry.version
7:                  else if entry.version > localVersionVector[entry.nodeId].version then
8:                      localVersionVector[entry.nodeId] ← entry.version
9:      return localVersionVector
```

3.6 Replica Migration

In order to provide fault tolerance in our system, the replicas position in the overlay has to be able to be adapted, more specifically, in the case when a node fails all the direct child nodes have to reconnect to a different node.

As we want to preserve the causality guarantees, the child nodes will try to replace the previous seed node with a node that is within the visible radius and that has a lower depth in the global overlay, first it will try to connect to the seed node of the previous seed node, and so on, until it successfully establishes a connection or it tries all known nodes without success, in which case the node cannot recover from the failure until one of known nodes recovers first.

When connecting to the new seed node we have to ensure that both nodes agree over the view they have of the system, for this purpose when changing seed node, a synchronization protocol, presented in the Algorithms 3.5 and 3.6, needs to be executed,

where the state of both nodes is compared and they exchange commands in case either is detected to have executed commands the other does not know about.

During the migration process the node can only execute local read and write operations, though the write operations from clients that require a certain number of confirmations(this will be detailed in the next section) will not be completed until the migration process terminates.

3.7 Client

As mentioned before, the client has its own component because it is an integral part of the system, due to the fact that some logic needs to be executed on the client side and it needs to maintain state.

When connecting to a node, an handshake protocol needs to executed with the server before any operation executed in order to ensure that server is in a safe for the client to join without breaking causality.

This handshake consists in the client sending the metadata that he received on the most recent response from the server, which will use it to check if the receiving replica is at least as up to date as the state represented in the metadata sent by the client, it might not be because the messages propagating the operations have not reached him yet, in which case the client cannot start executing operations on that server and must wait until it is safe.

In case the client does not have any metadata yet, which will happen when it is connecting for the first time or is starting a new session, it will always be safe to connect to the server, so the client can skip the handshake step if it has no metadata stored locally in its client library.

Each client can configure the number of acks that must be received before receiving the response for write and delete operations, this value can only range from 0 up to maximum distance that determines the radius of known nodes. This property has a direct impact on how likely other nodes already contain the operations executed by the client, which is required in order to migrate to another node without breaking the causality guarantee.

3.7.1 Metadata

In each response the server appends metadata with a partial view of the system that the he currently has, it only contains information about current node and the nodes for whom the current node is within reach by the maximum distance that is configured (the seed node of the current node seed node, and so on). The sub nodes are not included.

The metadata sent by the server contains the following : the version vector entry for each of the nodes along with the respective node id and how far each node is from the root of the tree overlay(i.e., the cloud). The latter is required to sort the known nodes into a list in descending order, this list is then iterated from the beginning until a connection

Algorithm 3.5 Synchronization Protocol - Part 1/2

Interfaces:
getCommandKey(command)
vvMerge(versionVector) ▷ Defined in the algorithm 3.4
applyCommand(command)
State:
seedNode

1: **procedure** HANDLEJOINREQUEST($sender, nodeId, versionVector, commands$)
2: **for all** $command \in commands$ **do**
3: $applythecommandsandupdatetheversionvector$
4: **if** $nodeId \notin localVersionVector$ **then**
5: **if** $localNodeId \notin versionVector$ OR $localVersionVector[nodeId] = versionVector[nodeId]$ **then**
6: $completeJoinRequest(sender, nodeId, versionVector, OK, \emptyset)$
7: **else if** $localVersionVector[nodeId] < versionVector[nodeId]$ **then**
8: $sendJoinResponse(sender, localNodeId, localVersionVector,$
9: $NeedMissingCommands, \emptyset)$
10: **else if** $localVersionVector[localNodeId] = versionVector[localNodeId]$ **then**
11: **if** $localVersionVector[nodeId] < versionVector[nodeId]$ **then**
12: $sendJoinResponse(sender, localNodeId, localVersionVector,$
13: $NeedMissingCommands, \emptyset)$
14: **else if** $localVersionVector[nodeId] = versionVector[nodeId]$ **then**
15: $completeJoinRequest(sender, nodeId, versionVector, OK, \emptyset)$
16: **else if** $localVersionVector[localNodeId] > versionVector[localNodeId]$ **then**
17: **if** $localVersionVector[nodeId] < versionVector[nodeId]$ **then**
18: $sendJoinResponse(sender, localNodeId, localVersionVector,$
19: $NeedMissingCommands, \emptyset)$
20: **else if** $localVersionVector[nodeId] = versionVector[nodeId]$ **then**
21: $version \leftarrow versionVector[localNodeId]$
22: $newCommands \leftarrow getCommandsAfter(version)$
23: $sendJoinResponse(sender, localNodeId, localVersionVector,$
24: $OkWithMissingCommands, newCommands)$
25:
26: **procedure** COMPLETEJOINREQUEST($sender, nodeId, versionVector, status, commands$)
27: $vvMerge(versionVector)$
28: $sendJoinResponse(sender, localNodeId, localVersionVector, status, commands)$
29: $subnodes \leftarrow subnodes \bigcup nodeId$
30: **if** $seedNode \neq \emptyset$ **then**
31: $send(seedNode, NEW\_SUB\_NODE, nodeId, localNodeDistance + 1)$
32: **procedure** RECEIVE($NEW\_SUB\_NODE, nodeId, distance$)
33: **if** $|distance - localNodeDistance| < max\_distance$ **then**
34: $subnodes \leftarrow subnodes \bigcup nodeId$
35: **if** $seedNode \neq \emptyset$ **then**
36: $send(seedNode, NEW\_SUB\_NODE, nodeId, distance)$

Algorithm 3.6 Synchronization Protocol - Part 2/2

```
1:  procedure HANDLEJOINRESPONSE(nodeId, versionVector, status, commands)
2:      if status = OK then
3:          localVersionVector ← vvMerge(versionVector)
4:      else if status = NeedMissingCommands then
5:          version ← versionVector[localNodeId]
6:          missingCommands ← getCommandsAfter(version)
7:          send(JOIN_REQUEST, localVersionVector, localNodeId, missingCommands)
8:      else if status = OkWithMissingCommands then
9:          for all command ∈ commands do
10:             key ← getCommandKey(command)
11:             if key ∈ localKeys then
12:                 applyCommand(command)
13:                 propagateCommand(command, subscriptions[key])
14:         localVersionVector ← vvMerge(versionVector)
```

can be successfully established. This way we ensure that we always connect to the closest possible node, which is ideal since the chance of a node being in a safe state for the client to join decreases as the distance from the previously connected node increases (inversely proportional).

3.8 Summary

In this chapter we defined the system model upon which we built, followed by defining the overlay structure in which the nodes are organized. introduced and detailed the proposed algorithms.

In the next chapter we will provide details about the implementation so that we can evaluate and test the proposed algorithms proposed in this Chapter.

Chapter 4

Implementation

After having defined the algorithms that compose our solution, the next step consists in creating an implementation so that our solution can be properly evaluated and tested. In this chapter, we will start by explaining why our solution was built on top of Redis, in Section 4.1, followed by a detailed explanation of relevant parts of how Redis works in Section 4.2, complementing the introduction done in the Section 2.6.4. In Section 4.3 we present relevant implementation details, namely of the changes made to Redis when building our solution and the client library we implemented.

4.1 Datastore selection

Initially the plan was to use Cassandra as the starting point for our system due to its popularity and its built-in support for various consistency levels and replication strategies, which fit well within our goal of adding an additional replication strategy. However after starting to familiarizing with the code base it was determined that it would be too complicated to implement the desired changes due to the complex execution flow. Hence Redis was used as the alternative, it has excellent code documentation and a smaller code base, made understanding the execution flow an easier process, and by extension it was also simpler to adapt and extend to achieve the behaviour required by our solution. The programming language used to write Redis is the C programming language.

4.2 Redis Internals

This section contains relevant details about how Redis works internally which were not addressed in Section 2.6.4 but are important as our solution built upon these functionalities.

4.2.1 Event Loop

Redis implementation follows the event loop design pattern, with this design applications don't need to worry about concurrent as one event is processed at a time and the processing is done all on a single thread. Redis is built on top of an event loop framework where events are registered and a callback for the when the event is triggered is given. The callback defined for these events must execute non blocking code, it must check if there's any processing that can be done given what it is immediately available when the event is called and then it must hand back the control so that no time is spent waiting instead of processing other events in order to maximize the cpu usage. An event is typically associated with a file, and could be triggered when the file is either readable or writable. As with linux everything is file, this can be used to define what should be done when a socket from a client is readable, possibly containing a command to be executed. The event callback would then read what was available without blocking and if a full command was read then execute it, otherwise hand back the control and wait for the next time the callback was triggered where it reads some more, this process is repeated until the whole command was received and then finally it would be processed without having wasted time waiting the full command to be read in one go. This paradigm has the great advantage that there is no need to worry about concurrency, greatly simplifying the programming logic as there is no need to synchronize the access to the program state.

4.2.2 Command Execution

Redis follows an execution flow that is very similar to the one seen in REST web servers, though instead of defining endpoints with URLs, it takes a more simple approach where a function is associated with a command name. Adding a new command is as simple adding an entry to the hard coded array containing the command mapping, then during start up redis populates a key value hash map with the command names and functions in order to efficiently find the command given the name.

When a command is received by the server it goes through a series of functions that populate the required structures with the request data and perform multiple checks before the command is actually executed, as well as executing some actions after its execution, such as command replication, key expiration, etc. This is very similar to execution flow present in REST frameworks, where each request goes through a series of filters before and after the actual function that handles the request path is executed.

As this is a common flow, and specially because it does not use complex techniques such as dependency injection or aspect-oriented programming which are more commonly used in modern programming languages, it was relatively easy to understand the execution flow just by analysing the code.

4.2.3 Replication and Persistence

The default configuration on Redis doesn't provide any replication or persistence as it is an in-memory database, that is, the whole dataset is always loaded into memory. However it can be configured to persist the data into the disk but this is always done in the background in an asynchronous way. As for the replication, it is possible to configure a node to be a replica of other node, but the replication is always done asynchronously, meaning that the user receives the response before the replica node confirms that the data was received.

4.2.4 Key Expiration

When setting a key it is possible to indicate a period for which the key will be valid, after which it will be as if the key was deleted. Keys with an expire time are also stored in an extra dictionary, besides the main one where all the keys are stored. This simplifies the expiration procedure and optimizes it by reducing the keys that have to be checked when expiring keys, which is done by iterating over the dictionary that contains the keys with the expire time set. The key expiration is done in both an actively and passively. The active part is done by periodically by sampling random keys from the dictionary and checking if their expiry time has passed. The passive part is when a read command is executed, before returning the key to the client a check is done on the key expiration time to see if it has already passed, in case it did then key is deleted and its value is not returned to the client.

4.2.5 Eviction of keys

Redis can be configured to evict keys when the occupied memory goes over a configurable limit. It offers two strategies, LRU and LFU, which are implemented using a probabilistic solution, due to the fact that checking comparing all the keys has a high performance cost and this is an operation that might be executed often so having good performance is a must. By default these strategies can be used to evict keys from the whole key set or only from keys which have an expiration date.

4.3 Prototype Implementation

In this section, we present relevant implementation details in our system, mainly focusing on how we extended redis while leveraging the functionalities already implemented as much as possible .

The communication protocol between the nodes was built on top of the RESP protocol. Support for reading and processing more than one message in one event trigger was implemented in order to improve performance by reducing the number of I/O operations, which have a major impact on the server performance.

The version vector serialization was implemented using mmap, [1] this approach was considered the best because it automatically handles the persistence of the data and we get a memory address (i.e., pointer) which can then be accessed and written to as a normal memory allocation in the code. reducing complexity as we do not need to write logic to synchronize the in-memory changes to the persistent file system nor be aware that the data is being persisted when writing.

4.3.1 Communication Links

The nodes communicate using two links, one is only used by the sub node to execute read operations on the seed node, while the other can be used by both nodes to send protocol messages. On the first link the communication is done following the standard redis protocol and the sub node behaves as just any other client connected to seed node, the only difference is that when the connection is established a node executes a command to identify himself to the seed node, which then marks the connection with that node id so that when a read command is executed for a key that exists, that sub node is added to the subscriptions for that key.

As for the second link, opposite to the first where the interaction was always started by the sub node, this one can be used by both nodes to send messages asynchronously between them. This connection is also marked with the id of the node at the other end and its processing is done using the event loop and registering a read event handler to process the messages received. The messages protocol we defined is built on top of RESP [2] as it provided all we needed and it there was also the advantage that support for is was already built into redis, meaning we did not need to worry about the low level aspects serialization.

It is possible that enough data is readily available when the read event is triggered so that it contains several complete messages, this allows processing multiple messages in one go, reducing the number of I/O events, hence improving the overall performance. As an extra optimization we only send the status update message to the sender after processing all the available replies, this avoids sending update messages for the intermediate updates that the server goes through when processing all the messages.

4.3.2 Command List Serialization

The command list contains not only the executed command, but also a copy of the resulting version vector after executing the command, along with the command id. This data needs to be kept while the commands are not acknowledged by all the nearby nodes. For this reason besides keeping the list in memory, the commands are also written to persistent storage, though the server does not wait for the commands to be written in the disk before replying to the client, this could be implemented but would cause considerable

[1] https://www.man7.org/linux/man-pages/man2/mmap.2.html
[2] https://redis.io/topics/protocol

impact on the performance, instead the file contents are flushed to disk periodically in the background. The command list serialization was implemented by using a rolling file strategy, similar to what is typically used by logging frameworks to avoid creating an ever growing file, where after a certain amount of commands a new file is created. Using this strategy allowed the cleanup for commands that had been marked as persistent to be easily implemented, as soon as the last command in a file was marked as persisted, the file could just be deleted. When loading from the disk on startup all the files in the directory would be loaded and the already confirmed commands would be ignored. This is a minor disadvantage but we can live with as the extra disk space taken up by the already confirmed commands is considerably small by today's standards. The files where the commands are stored always contain a fixed size index (possible because the number of command per file is not dynamic) at the start, which allows immediately seeking to the offset where the desired command is located, meaning there is no need to read the whole file just to find a command.

4.3.3 Optimizing conflict detection

When a concurrent operation is detected, there is a conflict if the value for the key in the concurrent command was changed in one of the concurrent operations, in which case we are using a last write wins policy, requiring that the command applying the new value is also propagated back to the sender in order to overwrite the previous operation as well as to all the other required nodes. Our first implementation operation iterated over the list of executed commands, starting from the most recent one up to the a command that was outside of the concurrent commands interval, the it would parse the command to determine the key present and finally compare it against the key in the current command. Although this worked fine initially as a proof of concept, it was not a very optimal solution and as redis is incredibly optimized to run as fast as possible this needed improvements. The solution we ended up implementing avoids iterating over the list of commands by keeping track of the last time each local key was updated and the node that propagated the command, this way we can determine if there is a conflict if the last time the key is within the interval of concurrent commands and the id of the node who caused the last update is different than the node who propagated the current command. As an extra optimization, that saves both memory and CPU cycles, we do not store and compare the actual id of the nodes but instead use an hash generated using the SipHash [5] function, support for it was already included in redis and the result size is only 64 bits, which when represented as a 64 bit integer is considerably faster to compare than a string.

4.3.4 Dynamic Data Placement

All edge nodes are initialized with an empty dictionary that will store information about current local keys. A connection with the seed node is also established that will be used solely to fetch keys. The edge node sends a special command through this connection

that flags as a fetch connection, the command also contains the id of the node so that the seed node can know to whom this connection belongs to, this is useful because when a read command is executed through this connection, it adds the sender of the request to the list of active subscriptions in case the key exists, this way the nodes do not need send an extra message just to subscribe to a key and execute a read command to get the key value separately.

Each local key contains a list of subscribed nodes that are currently interested in receiving updates for that key, allowing the nodes to only receive updates over relevant keys, enabling partial replication.

When a key is deleted, it is removed from the local keys dictionary, the list of active subscriptions is also deleted as keeping a list over a non existing key serves no purpose and takes up memory. If sometime in the future the key is added again, then the list of subscriptions will start by being empty. In case the sub nodes are still interested they will eventually make the fetch request again and the list will be re-populated over time.

4.3.4.1 Command Interception

In order to verify if the key indicated in a command is a local key we have to intercept the command before it is actually executed. After analysing the code we were able to insert a function that is called before the command is executed, in this function is where we verify the type of command, if the key in the command is local key and decide if we need to send a read request to the seed node before the command can be executed. If a read request needs to be sent to the seed node it is done in asynchronously in order to not block the server, this required saving the current client context, stopping the current command execution and allowing the server to keep processing requests for other clients, once the reply from seed node is received the client context is restored, the command execution resumes at the point it was paused, and finally the client context saved temporarily is cleared. Implementing this required adapting redis to allow stopping and resuming the command execution, as it was clearly not built with this in mind.

4.3.5 Replication

After an operation is executed locally, it may have to be replicated to other nodes. In case the current node is an edge node, the command always has to be replicated to the seed node. It must also be propagated down to all the nodes included in the active subscriptions for the key present in the command.

The replication is done asynchronously in the background. The client may choose to set a configurable parameter that determines how further up the node tree a command must be replicated until it receives the response.

As replication is only done after the command is already executed, meaning the reply was already written to the client buffer or socket by default, some changes in the way redis works were required in order to implement our replication protocol. First, we check if the

client has set the number of acks he desires to receive to a number greater than zero, if yes then we flag the client with a special value that will indicate to the functions that write into the client buffer to write into a new temporary buffer instead of the client output buffer as is normal. After the command is executed we save the client, the command response and the command ID into an object which is then added to a list of blocked clients. When the required acks for a request are received the response is finally written into the correct client output buffer and the client is removed from the list.

4.3.6 Autonomous replicas removal on edge nodes

This component was built on top of the internal components responsible for the key expiration and for the eviction of keys. The goal was to allow only the removal of keys in a safe way and periodically remove unused keys, while adapting to the available resources in the system. In order to implement this functionality we leverage two functionalities already supported by redis, namely the key expiration and the key eviction. The whole process involves three main steps :

When removing a key subscription from a subscriber (i.e., sub node), if there are no more active subscribers that key then an expiration time is set for the key, if a subscriber is added for this key before the expiration time is reached then the expiration time is removed and the key will no longer be removed, the expiration time can be updated if there is a local operation (only from actual clients, not replicas) executed over the key, in which case the time is not removed, instead it is advanced, effectively delaying the key removal while it is in use (as long as operations over the key continue being executed before the expiration time is reached).

Once the key is expired then it is added to a dictionary from which keys are periodically picked in order to be evicted. Not removing the keys immediately will probably result in higher resource usage as a trade off for reducing the chance that a key is removed just to be added again a few moments, but if the resources are available then the trade off could be worth it. The periodic process that executed to remove keys adapts the number of keys it evicts based on the available resources, the lower the amount of memory available, the more aggressive it is.

4.3.6.1 Key Eviction

As presented before, the key eviction system is responsible for removing keys when the used memory surpasses a configurable limit. By default Redis already allows different strategies to perform this eviction, more importantly we could choose if we wanted to consider all the keys in the system for eviction or only the keys that had an expiration time set. We extended this by defining a new eviction strategy, where only keys that had been already expired would be considered for eviction, integrating with the changes made to the key expiration process as explained in the next section.

We extended the system to allow only keys that don't have any pending operations to be evicted. To do this we refactored the the process used by redis to randomly pick keys for eviction to receive an optional function that would validate whether a key could be removed or not. Then defined a function to check if a key contained pending operations and finally defined a new eviction policy to redis that would cause this function to be used to filter the keys and choose .

4.3.6.2 Key Expiration

As we want to remove unused keys on edge nodes after a certain time, it made sense to build our solution on top of the expiration functionality already built into redis. The first step is to set an expiration time, this is done initially when a new local key is added or when removing a subscription a sub node from the list of subscriptions for a key results in an empty list. Now if the key is accessed locally, be either a read or write operation, the expiration time is advanced, so that keys that are being used actively are not removed. Originally redis simply removed the expired keys, however we changed the process so that instead of removing the keys immediately after this period, they are added to a set of keys that will be considered for eviction. For this reason an extra check needs to be done when adding a new subscription for a key that previously did not have any, it is possible that the key had already expired but even though it had not yet been evicted, in which case the key must also be removed from set of expired keys.

4.3.7 Client

We created a client library for our version of Redis based on a library that offered support for the RESP protocol instead of using an existing client such as Jedis, which has more features than we need. By creating our own client from nearly zero it is easier to build it to our needs instead of adapting an existing code base with many features that we do not need. This was feasible because our client only needs to support three simple commands which were already part of redis (get, delete, and set) and a special command that identifies the client as and edge client which will allow him to receive metadata with each response that he gets. We did have to re-implement some of the logic already present in Jedis to support redis cluster mode but that was easily done, it does not involve much logic besides checking if a node is in cluster mode and if yes then process the cluster configuration and establish a connection with each node and redirect the requests to the appropriate node based on the key has.

4.3.7.1 Enforcing causality

As mentioned before, with each response there is also some metadata that is sent. The processing of the metadata is made transparent by the client library. It comes as an extra field on the reply alongside the actual response to the command issued, the client library

extracts both the metadata, which is stored locally by the library, and the command response which is then delivered through the library public interface.

When a replica fails the client will try to migrate to another replica in a safe way. It iterates over the known nodes which are sorted by proximity to the node we were connected to. Upon picking the node we want to connect to, a query for its network location given its id is sent to the discovery service. After obtaining the network location it will try to establish a connection followed executing a special command containing the most recent metadata stored locally and the id of the node it was connected to, so that the server can indicate whether it can start executing operations without breaking causality.

Configuring commands confirmations: The number of *acks* (a confirmation that a node has received the replicated command) received by the connected replica from its seed nodes required to confirm operations is configurable per client connection. This allows the applications to adjust the required *acks* in the way that fits their requirements. A greater number of confirmations increases the chance of handling a fault without breaking causality but has a direct impact on the system performance.

4.4 Summary

In this chapter we explained the modifications required to Redis in order to build our solution, detailing how the new functionalities were built on top of functionalities already provided by redis when possible.

In the next chapter we will evaluate the implementation and analyze the results obtained.

Chapter 5

Evaluation

In this chapter we discuss how the evaluation of our implementation was done and analyze the results. We start by defining the experimental setup in Section 5.1, followed by a definition of the experiments done in Section 5.2 and finally we present and analyse the results in Section 5.3

5.1 Experimental Setup

We used the Grid5000 tested to run the experiments, more specifically the *gros* cluster available at the *Nancy* site. All the experiments were run on the same cluster, where the machines have the following configuration: an *Intel Xeon Gold 5220* with 18 cores, 96GiB of RAM, a 480GB SSD as the main drive and two network interfaces of 25 Gbps each.

As we want to simulate a network of connected nodes with more limited resources than what is available in a machine, we used Docker to limit the available CPU and RAM resources by launching one container per node. As a single node will only use a fraction of the total available resources in a machine, we launched up to virtual 10 nodes per machine.

We used *EnOSlib* [1] to automate the resources reservation and jobs creation on Grid5000, to configure a docker swarm containing all the hosts in the job and execute commands across all the hosts required to setup the nodes for the tests. This included loading required docker images and creating an overlay network connecting all hosts, so that containers running on different hosts could communicate transparently.

In order to simulate the clients using the application we used Yahoo! Cloud System Benchmark (YCSB) [8] to generated the workloads and perform the benchmarks. We limited the number of threads per machine to 1000, if more were required for an experiment

[1] https://discovery.gitlabpages.inria.fr/enoslib/

then multiple YCSB instances were launched on different machines.

We will simulate a network composed of 50 up to 100 edge nodes, split evenly between U.S.A. and Europe. There are also 10 cloud nodes located in Germany, configured in redis cluster mode, with 1 replica per node, meaning 5 out of the 10 cloud nodes are replicas. Each node can be uniquely identified by its type and id, where the id is simply the index of the node. Each of the edge nodes was assigned a geographic location, the locations were determined by picking 50 of the most populous cities in the U.S.A. and in Europe. We then needed to determine the connections between the nodes to build the overlay, to do this we applied the following algorithm: for each node, pick the closest node as seed node that does yet have three child nodes. After this setup we were ready to determine an approximate latency between the connected nodes We defined a latency map containing the latency between the IP's that needed to be applied in order to better simulate a real world scenario. In order to apply the latency between the nodes correctly we needed to assign the IP of a redis instance in a deterministic way, depending on its id and the type of node in order to avoid conflicts. The cloud nodes were assigned IP's in the 192.168.2.0/24 sub network, while the edge nodes were assigned IP's in the 192.168.3.0/24 sub network. The last 8 bits of the IP were assigned with the value of the node id.

The latency between the nodes was simulated using the TC[2] utility available in Linux systems, applying the commands to the network interface inside the docker container. The latency map was included in the docker image along with a script that applied the TC commands to simulate the required latency based on id and type assigned to the current container, after the commands were applied the script did nothing else besides executing the originally requested command passed as parameter. This is how the latency was applied either on the redis containers for the edge nodes as well as for the YCSB containers during the baseline test.

The available resources in the edge nodes depend on the depth of the position where the node is located in the generated overlay, we adapted the resources using the following quotas, as was done in the work presented in [9], and are presented in Table 5.1.

Depth	Quota
0	1/2
1	1/3
2	1/4
3	1/5
4	1/6

Table 5.1: Resources quota per depth

[2]https://linux.die.net/man/8/tc

5.2 Experimental parameters

We split the experiments in two groups in order to reduce the number of experiments that we needed to execute so that it could be done or we would have needed hundreds of hours to do them all, The configurable parameters in our experiments that we will test are the following:

1. **Number of edge nodes :** The number of edge nodes in the system, when less than 100 nodes are used in an experiment, the nodes that are excluded are the ones closest to the bottom of the overlay, furthest from the cloud.

2. **Number of acknowledgments:** This parameter defines the number of confirmations an operation must receive before the response is sent to the client, hence has a direct impact on how long each command takes to execute. The number of confirmations is defined as the number of seed nodes that the connected node is sure that have received the operation.

3. **Metadata Distance / Radius :** Defines the size of the view for the edge nodes, having an impact on the number of update messages that are sent as the bigger the view the more entries in the version vector need to be maintained and the further an update needs to be sent.

4. **Read / Write Operations ratio :** The ratio between the write and read operations executed by the clients, configured in YCSB.

5. **Number of clients :** The total number of client threads launched by YCSB

In the first group of experiments our goal was to evaluate the impact of key parameters in the performance of our system, here we varied the number of acknowledgements required to confirm the operations, the ratio between read and write operations, the number of edge nodes in the system and the radius that defines the size of the view for the nodes. The values for these parameters and the ones that were fixed in this group of experiments are detailed in Table 5.2.

In the second group of experiments our goal was to evaluate how the system scaled, so we tested with various number of clients. The values used are detailed in Table 5.3.

Parameter	Values
Acks	0, 1, 2
Number of edge nodes	50, 75, 100
Metadata Distance (MD)	1, 2
Read / Write Operations ratio	0/100, 50/50, 95/5
Number of clients	2500

Table 5.2: Configuration for group 1 of experiments

Parameter	Values
Acks	1
Number of edge nodes	100
Metadata Distance (MD)	1
Read / Write Operations ratio	50/50
Number of clients	100, 300, 500, 1000, 2500, 10000

Table 5.3: Configuration for group 2 of experiments

5.2.1 Baseline

As a baseline we will use the unmodified Redis datastore configured in cluster mode with asynchronous replication, with 1 replica per node, as done in the other groups of experiments. The consistency guarantee provided by this configuration is eventual consistency, as operations are replicated in the background it is possible for a client to read an outdated value from the secondary node after executing a write on the primary node.

In these experiments the system's total resources will be equivalent to the total resources used by running the 100 edge nodes in the defined overlay plus the 10 cloud nodes. As there are no edge nodes the parameters for the number of edge nodes, number of acknowledgments and the radius size are not applied in this set of experiments. In table 5.4 we present the values intervals for the parameters in this set of experiments.

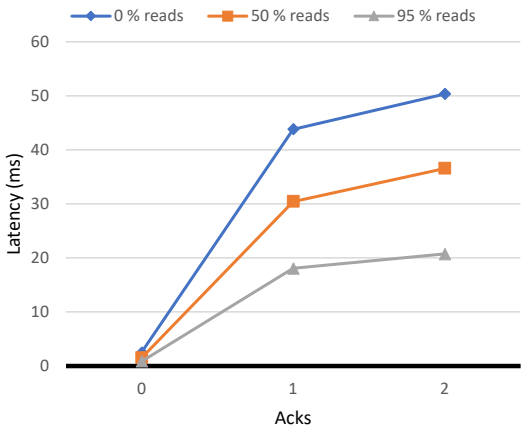
Parameter	Values
Read / Write Operations ratio	50/50
Number of clients	100, 300, 500, 1000, 2500, 10000

Table 5.4: Configuration for the baseline experiments

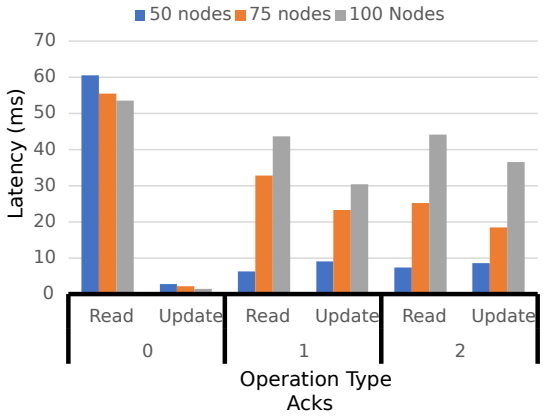
5.3 Results

5.3.1 Multiple configurations and the impact of various parameters in the system

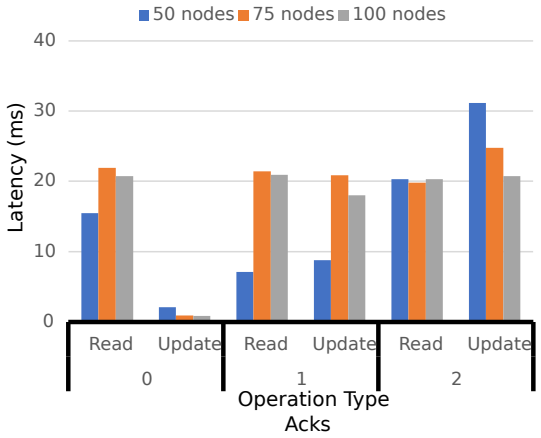
In Figures 5.1a, 5.1b, 5.1c we report the results of the impact of increasing the number of required acks on user perceived latency across different read/write operations ratio scenarios. In Figure 5.1a we can see that the latency for update operations increases for all the read/write ratios as the required acks increase, however the increase is greater when we the ratio for write operations is greater, which would be expected as more operations that need confirmation before the client receives a reply are executed. In Figures 5.1b and 5.1c we present the comparisons between the latency for update and read operations, as well as how the latency changes for each operation with the increase of required confirmations. The difference seen between the latency in both operations when no confirmations are required can be explained by the fact that when a read operation



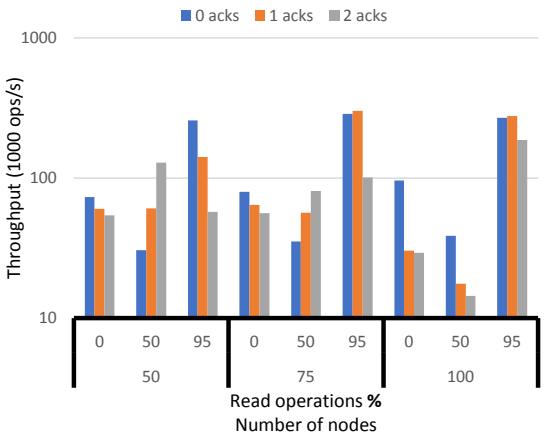
(a) Latency for update operations



(b) Latency per operation with 50% reads



(c) Latency per operation with 95% reads



(d) Throughput

Figure 5.1: Performance impact for multiple acks configurations, with 2500 clients and MD=1

is executed over a non local key, the read request is propagated up the overlay until the key is found or the key is determined to not exist, this takes a considerable amount of time when compared to a command execution that can be completed without having to exchange messages with any other node before being able to send the reply to the client. This difference is no longer present when the required confirmations is increase to either one or two, here we observe approximate values for both latency's in Figure 5.1b independent of the number of confirmations, however in Figure 5.1c we do see a considerable difference, specially on the scenario with 50 edge nodes (as well as a smaller one on the scenario with 75 edge nodes) and two confirmations required.

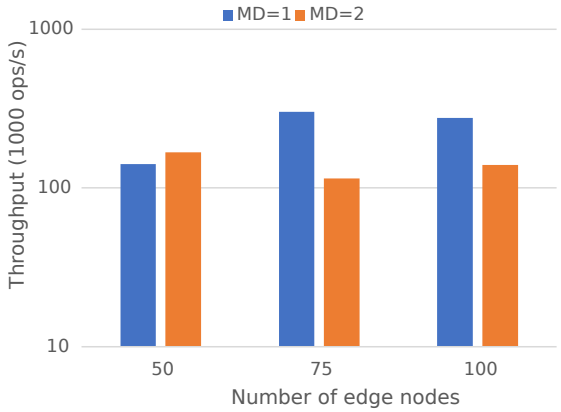
The higher latency observed in some cases when the number of nodes increases may be caused by the higher number of messages that are being exchanged by the nodes. And for the read operations the increase may also be due to the fact that more nodes may need to be contacted in order to obtain a key value as more nodes are in the system, the nodes

that are added are always at the bottom of the overlay, meaning they have nodes between them and the cloud.

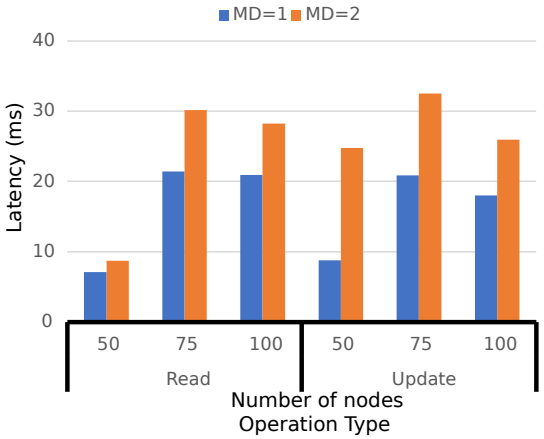
Acks: Figure 5.1d show the impact of changing the number of confirmations on the throughput is represented on multiple scenarios for the number of nodes and the ratio of read/write operations. We can observes that in the scenario with 100% write operations, the throughput is affected by the increase in required confirmations, as would be expected, independently of the number of nodes in the system. On the scenario with the ratio as 50% reads/50% writes, the observed behaviour differs, the throughput grows when the number of nodes is 50 and 75, however in decreases when there 100 nodes. The growth observed here seems counter intuitive at first, since we are increasing the required confirmations for an operation it would be acceptable for the throughput to decrease, however it does not decrease in all cases. A possible explanation would be that because write operations start to take longer, there are less messages being propagated in the system, which in turns leaves more free resources to process the read operations, hence increasing the throughput, even though write operations will take longer for the clients doing them. When the system has 100 nodes, we finally observe a decrease in the throughput, due to the higher number of nodes there is an increase in messages flowing through the system and read operations will also take longer to execute as explained previously, this seems to bring up the resource usage, consuming the resources that would become available with the reduction of write operations, making the performance impact of increasing the confirmations clearly visible. for the 95% read operations scenarios, we consistently see a drop in the throughput when increasing the required confirmations, probably due to the explained previously reason that causes the performance drop in the scenario with 100 nodes and 50% read operations.

Metadata Distance: In Figure 5.2a we can observe that increasing the distance metadata is propagated does not have a considerable impact when the system contains just 50 nodes, we even see a slight increase in performance when changing the value from 1 to 2, a possible cause is that the increase in update messages allows the operations to be confirmed faster but does not increase significantly as to saturate the nodes due to the low number of nodes in the system. With the increase of number of nodes in the system the performance impact becomes greater, as the combination of adding more nodes and increasing the metadata distance starts to saturate the system. However the penalty is greater where there 75 nodes than when there are 100 nodes, this may be cause due to how the overlay is defined, changing how the load is distributed (nodes with more child nodes have more messages to process)

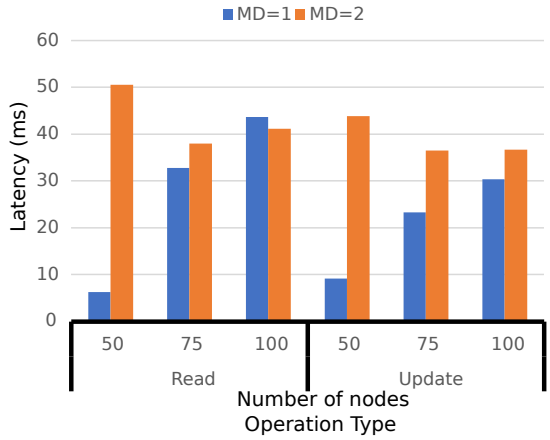
We can also see in Figure 5.2b, 5.2c and 5.2d we show that increasing the metadata distance consistently affects the latency, both in read and update operations, independently of the read write ratio. The greatest impact is seen when there is just 50 nodes in



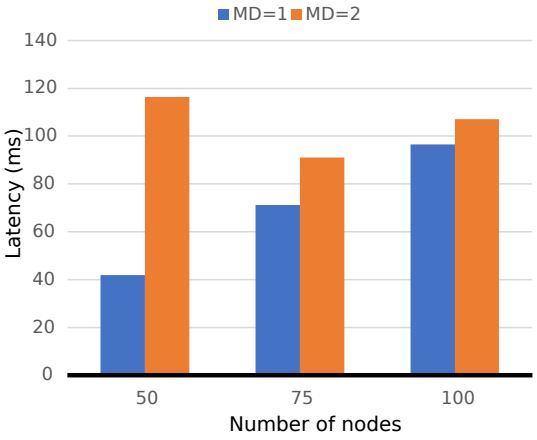
(a) Throughput



(b) Latency per operation with 95% reads



(c) Latency per operation with 50% reads



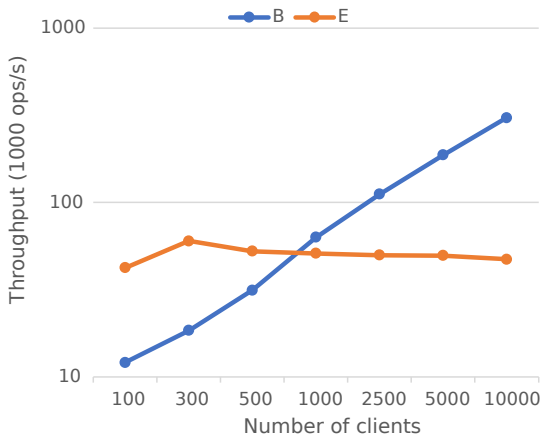
(d) Latency for update operations

Figure 5.2: Performance impact for multiple Metadata Distance configurations, with acks=1

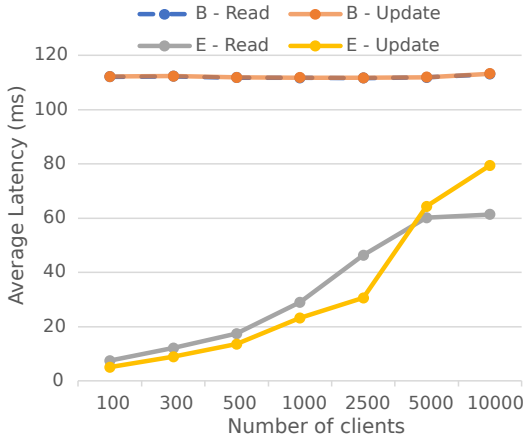
the system, indicating that there is not enough resources to absorb the impact hence the system's limit is reached more easily.

5.3.2 Baseline

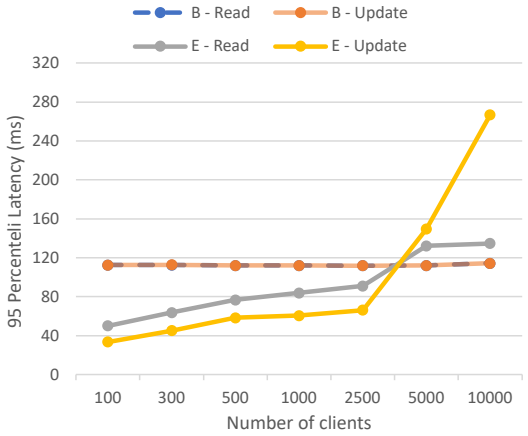
In Figure 5.3a we analyze how the system scales the throughput while increasing the number of client threads in YCSB and compare it with the baseline. The throughput is greater in our system until the 1000 client threads, the performance limit seems to be achieved around the 300 clients, after which the throughput only decreases while in the baseline it doesn't stop improving, we only tested up to 10000 client threads but this was not enough to saturate the baseline system. There is no doubt that our system is easily outperformed in terms of throughput when comparing to the unmodified redis, however this is not surprising after having looked at how redis is implemented, everything is highly optimized, so much that naively adding any extra logic to the server results in a huge



(a) Throughput



(b) Average latency per operation type



(c) 95 Percentile latency per operation type

Figure 5.3: (B)aseline vs (E)dge

performance hit. We did optimize the implementation as much as possible for the extra functionalities we needed to implement but there's simply no way around the fact that we are adding extra processing that is done for each command as well as simultaneously propagating commands across the nodes, which causes a command to be executed on multiple nodes.

Figures 5.3b and 5.3c report the 95 percentile latency and average for both read and update operations. In the baseline system the latency for read and write operations identical, where as in our system we observe that read operations have a higher latency, up until the 5000 client threads, where the update latency is nearly the same as read operations, but with 10000 client threads it increases significantly while the latency for read operations remains nearly the same. Our system offers a lower average latency overall for both operations when compared to the baseline, as denoted in Figure 5.3c, however the latency can be lower on baseline in the scenarios with 5000 and 10000 client threads when looking at the 95 percentile latency, seen in Figure 5.3b.

5.4 Summary

In this chapter we explained how the evaluation of our implementation was conducted, present the results and discuss them.

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

Chapter 6

Conclusion and future work

6.1 Conclusion

The edge computing paradigm is still recent but the problems it aims to solve are very appealing, namely reducing latency for the end user and better load distribution for distributed systems, by moving computational resources outside data centres, which concentrate the resources used by users in a relatively large geographic area, this is fine by now but with the expected growth of the rate at which data is generated it will stop being feasible to centralize the amount of resources required to process all the data in the area. The network infrastructure will eventually become a limiting factor as well if all the generated data needs to be sent to data centres.

In this thesis we explored an hybrid distributed data store that tries take advantage of the edge computing paradigm, it aimed at reducing latency by moving data closer to the edge while providing a causal consistency, which offers good performance and is easier to reason about the application state than the traditional weak consistency models. It still uses the cloud to store all the data to ensure better data durability as well as offering high availability, something that is not easily done at the edge..

In order to build our datastore we designed a replication protocol capable of dynamically moving data from the cloud to the edge. The presented protocol supported both genuine partial replication and causal consistency, the first is important in order to reduce the amount of metadata that needs to be propagated through the system while the latter offer a better programming model to build applications on top of the system, due to the evolution of the state of the application being easier to reason about. Our implementation that was built on top of redis (using the C programming language) offered a proof of concept, the implementation was considerably more challenging than expected, developing in a language where memory management falls upon the developer proved to be quite

challenging, .

The experimental results show that our system is capable of offering average lower latency and a 95 percentile latency lower in in some scenarios and that the throughput is considerably lower than in the original system, showing that there is room for improvement in our system. We also evaluated how our system adapts to the different possible configurations it can be used.

Regarding the initial goals of this work, we have achieved the main goal, which was to build a datastore that offers a dynamic replication protocol and causal consistency. Our solution is capable of offering lower latency but lacks in performance when analysing the throughput, hence it is possible that there is still room for improvement in our implementation and in our replication protocol. We did not get to explore the dynamic creation and removal of instances as well as a dynamic overlay that adapted to the system load as initially planned,

6.2 Future Work

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

Dynamic Overlay: In our solution we assume that the overlay membership is static, the resources available per node are static and the edges in overlay are predefined, we think that it is worth exploring solutions to dynamically restructure the overlay based on the system load and the available resources per node, as well as launch or remove instances as required.

Amount of metadata: Based on the experimental results, we think that there is room to improve the performance of our implementation and possibly in the replication protocol, which could possibly be achieved by reducing the amount of metadata exchanged by the nodes, both in number of messages sent and the payload of the message.

Conflict Resolution: In our solution we only implemented the last write wins policy to resolve conflicts, the current implementation could be extended to support more advanced strategies, such as CRDTs [26]

Bibliography

[1] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. "Volley: Automated Data Placement for Geo-Distributed Cloud Services." In: *NSDI*. 2010.

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

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

[4] A. Atrey, G. Van Seghbroeck, B. Volckaert, and F. De Turck. "Scalable data placement of data-intensive services in geo-distributed clouds." In: *CLOSER2018, the 8th International Conference on Cloud Computing and Services Science*. SCITEPRESS-Science and Technology Publications. 2018, pp. 497–508.

[5] J.-P. Aumasson and D. J. Bernstein. "SipHash: a fast short-input PRF." In: *International Conference on Cryptology in India*. Springer. 2012, pp. 489–508.

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

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

[8] 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*. 2010, pp. 143–154.

[9] P. Á. Costa, P. Fouto, and J. Leitão. "Overlay Networks for Edge Management." In: *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*. IEEE. 2020, pp. 1–10.

[10] Dormando. *Memcached*. URL: https://memcached.org/ (visited on 02/02/2020).

[11] Dormando. *Replacing the cache replacement algorithm in memcached*. 2018. URL: https://memcached.org/blog/modern-lru/ (visited on 02/05/2020).

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

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

[14] R. Guerraoui and A. Schiper. "Genuine atomic multicast in asynchronous distributed systems." In: *Theoretical Computer Science* 254.1 (2001), pp. 297 –316. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(99)00161-9. URL: http://www.sciencedirect.com/science/article/pii/S0304397599001619.

[15] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. "The Φ accrual failure detector." In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* IEEE. 2004, pp. 66–78.

[16] T. Hoff. "Facebook's memcached multiget hole: More machines!= more capacity." In: *High Scalability, October* (2009). URL: http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html.

[17] Y. Huang, Y. Shi, Z. Zhong, Y. Feng, J. Cheng, J. Li, H. Fan, C. Li, T. Guan, and J. Zhou. "Yugong: geo-distributed data and job placement at scale." In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 2155–2169.

[18] K. Kritikos. "Towards Dynamic and Optimal Big Data Placement." In: *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE. 2018, pp. 1730–1737.

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

[20] L. Lamport. "Time, clocks, and the ordering of events in a distributed system." In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 179–196.

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

[22] J. Leitão, P. Á. Costa, M. C. Gomes, and N. Preguiça. "Towards Enabling Novel Edge-Enabled Applications." In: *arXiv preprint arXiv:1805.06989* (2018).

[23] C. Liu, K. Ouyang, X. Chu, H. Liu, and Y. Leung. "R-memcached: A reliable in-memory cache for big key-value stores." In: *Tsinghua Science and Technology* 20.6 (2015), pp. 560–573. ISSN: 1007-0214. DOI: 10.1109/TST.2015.7349928.

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

[25] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. "Scaling Memcache at Facebook." In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 385–398. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

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

[27] S. Sivasubramanian. "Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service." In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: ACM, 2012, pp. 729–730. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213945. URL: http://doi.acm.org/10.1145/2213836.2213945.

[28] R. Van Renesse, D. Dumitriu, V. Gough, and C. Thomas. "Efficient reconciliation and flow control for anti-entropy protocols." In: *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. 2008, pp. 1–7.

[29] R. Van Renesse and F. B. Schneider. "Chain Replication for Supporting High Throughput and Availability." In: *OSDI*. Vol. 4. 91–104. 2004.

[30] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos. "ENORM: A Framework For Edge NOde Resource Management." In: *IEEE Transactions on Services Computing* (2017), pp. 1–1. ISSN: 2372-0204. DOI: 10.1109/TSC.2017.2753775.

[31] B. Yu and J. Pan. "Location-aware associated data placement for geo-distributed data-intensive applications." In: *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE. 2015, pp. 603–611.

[32] B. Yu and J. Pan. "Sketch-based data placement among geo-distributed datacenters for cloud storages." In: *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE. 2016, pp. 1–9.

[33] B. Yu and J. Pan. "A framework of hypergraph-based data placement among geo-distributed datacenters." In: *IEEE Transactions on Services Computing* (2017).

[34] S. Zhang, C. Liu, Y. Han, and X. Li. "Seamless Integration of Cloud and Edge with a Service-Based Approach." In: *2018 IEEE International Conference on Web Services (ICWS)*. 2018, pp. 155–162. DOI: 10.1109/ICWS.2018.00027.

[35] J. Zhou, J. Fan, J. Jia, B. Cheng, and Z. Liu. "Optimizing cost for geo-distributed storage systems in online social networks." In: *Journal of computational science* 26 (2018), pp. 363–374.