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

February, 2020

The work reported in this document was partially supported by
Fundação para a Ciência e Tecnologia through the Project NG-STORAGE (PTDC/CCI-INF/32038/2017)

Abstract

Nowadays the Cloud Computing model is the most widely used to support the operation of large scale distributed systems. However, it 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 to improve their users experience.

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 use both models. This however, raises the question on how to create a system that integrates both models to take the best of both worlds. As the edge is composed of a multitude of devices, each with different characteristics where the resources available may be dynamic, it presents a very different environment when compared to the one seen in data centres where everything is controlled.

In this work we will focus on a particular 2nd 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 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 para melhorar a qualidade do serviço dos seus clientes.

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 ambos os paradigmas seriam utilizados, o que leva ao problema de como criar um sistema que integre ambos paradigmas e aproveite o melhor dos dois mundos. Devido à Berma ser composta por uma grande variedade de dispositivos, cada um com caraterísticas diferentes e os recursos disponíveis podem ser dinâmicos, o que leva a que o ambiente na Berma seja consideravelmente diferente quando comparado com a Nuvem, onde se trata de um ambiente 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

Acronyms

CDN	Content Distribution Network.
DHT	Distributed Hash Table.
IoT	Internet of Things.
LRU	Least Recently Used.
QoS	Quality of Service.
SMR	State Machine Replication.
WAN	Wide Area Network.

Chapter 1

Introduction

1.1 Context

The Cloud Computing paradigm appeared not long ago, proposing the sharing of 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 capable of handling automatic scaling and fault tolerance that can be adjusted to the application's needs. 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 popularity, possibly leading to two undesired scenarios. Over provisioning resources will 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 to handle an unexpected surge in demand. The Cloud avoided these scenarios by offering a "pay as you go"pricing for their services that would 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.

This model works well enough with the typical web application where the user interaction is very limited, where the data flow is mostly uni-directional (server to client), with an occasional submission by the client. However, it does have 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, it is short-lived, meaning that is is only relevant for a small period and needs to be processed before it becomes irrelevant, and because it will reach a rate 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.

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 possibly reduce 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 accessed and produced.

1.2 Motivation

Edge Computing is still quite new and its capabilities have not been fully explored yet. It aims to solve 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 needed the most, 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 already exists on data placement for geo-distributed systems, such as [**volley**, **yu2015location**, **yu2016sketch**, **huang2019yugong**, **yu2017framework**, **zhou2018optimizing**, **atrey2018scalable**, **kritikos2018towards**], 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 patterns, which is critical when we are limited by the resources available on the edge and do not want to waste storage space with data that is no longer required.

1.3 Problem definition

There has already been some research in optimizing data placement, as well as in integrating the edge with the cloud (by processing the data before sending it, not storing data). However, we aim to take advantage of the edge to store and modify data as well, instead of just doing computations over generated data that is then sent to the cloud. To do so we need the following :

1. A data placement scheme capable of producing a location mapping for the data, maximizing co-locality, this is storing data that is frequently accessed together in the same nodes, while keeping the costs down and acceptable low latency. It should be dynamic, reacting autonomously to changes in the data access patterns and

resource availability on the edge, indicating where replicas should be created or deleted.

2. A replication protocol with dynamic membership to add and remove replicas as required. Responsible for keeping the replicas up-to-date, handling conflicts, and ensuring data durability. Replicas on the Edge might disappear at any moment so extra care is required to ensure data durability on the Edge.

1.4 Expected Contributions

In this thesis, we plan to explore the design space of cloud/edge data placement schemes that will be naturally integrated into storage systems (such as variants of Cassandra, ChainReaction, or C3), that automatically can make decisions on the life-cycles of replicas located in the edge. Such a data placement mechanism will also manage the creation and decommission of replicas to ensure that data exposed to applications is consistent (given the consistency model being employed) and also that data is not lost in this process (i.e., ensuring durability).

1.5 Document organization

The document is organized as follows : In Chapter 2 we introduce what Edge computing is, various areas of work relevant to our project, present other works in those areas and discuss why they are relevant to us. In Chapter 3 we detail our proposed work, including the system model, proposed model, evaluation, and work plan

Chapter 2

Related work

In this chapter, we will discuss and introduce various relevant aspects 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 and 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[**leitao2018towards**]. 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 by [**8456344**], 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 the 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. As in the Cloud model, scaling might be required to adjust to demand and maintain the desired Quality of Service (QoS). In [**enorm**] 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, 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 on 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 are 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 very 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 distributed among the 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 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 of this problem.

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 more consumed resources like bandwidth and processing power to

process information about data that is not stored on the replica. Genuine Partial Replication [**genuinepartial**] is a subset 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).

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 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. Chain presented in [**chainreplication**], it allows it is based on primary backup that allowed better high throughput and availability while providing linearizability. The nodes 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, 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 its removed from the chain. Failures are supported as long as N-1 nodes are not failing concurrently.

Multi Master Backup: 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. It is usually not used in a strong consistency model as ordering all the operations globally when we have a considerate amount of replicas would have a big impact on system performance. It works better with weaker consistency models, which require less synchronization between replicas allowing for better scalability.

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, which can be achieved using State Machine Replication (SMR).

2.2.3 Replication for Edge Computing

Partial replication is particularly useful 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 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 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 we want to take the most advantage of having all these replicas to get good performance. The 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 costs. 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 have a high cost as well. 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 [**memcached**] 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 intelligent clients. 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 uses the Least Recently Used (LRU) [**lru**] 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 in the last case when that value expiry date is reached. Each key has a corresponding expiry date that is changed with every write operation.

2.3.1.1 Handling Failures

Fault tolerance for cache systems is a slightly different problem when compared to 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 in 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 correct solution would be to have clients redirect requests directly to the lower layer while the cache replica was down, 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.

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, the client can always fallback to the lower layer. This solution has two main problems. For memcached specifically it

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

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 hence they do not synchronize for performance reasons. Other limitation not specific to memcached but also applies due to using constant 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 [**fb_memcached**] 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 [**r_memcached**] 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 the web browsers do or through a CDN. Both options improve user-perceived latency. 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 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 services.

Memcached [**memcached**] performs well enough for small scale systems, offering high throughput. Although some limitations come to light when expanding the system across

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

various regions and a considerable amount of servers, these are successfully addressed in the work presented in [**fb_memcached**] 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 over the data and only read operations are possible. 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, per example when users interact with other users that are close and any significant latency is easily noticed, like Pokemon Go.

One thing cache has in common with our work is that resources are limited, 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, 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 on the replicas. A 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 [**brewer2000towards**, **gilbert2002brewer**] 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.

Usually, partition tolerance is a must-have property because it is unavoidable in large distributed systems. This leaves us with a choice between the other two. 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,

[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).

mainly No-SQL databases, like Dynamo [**dynamo**], Cassandra [**cassandra**], ChainReaction [**chainreaction**], and COPS [**cops**].

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 this cases is required on the application side. We will now talk about two 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 [lamport2019time] : In this model the values returned by a replica respect the causal order. 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. This is due to the fact that causal dependencies between operations must be tracked and the ones that are related ordered. Another advantage is that causally independent operations do not require an order, they can be executed concurrently which improves the system's performance.

Causal+ Consistency [cops] : Causal consistency does not order concurrent operations, in case these two operations are conflicting, replicas are allowed to diverge forever, returning two different values. 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 basic 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 eventual systems where old results might be returned or in casual 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(i.e., using Lamport Clocks [**lamport2019time**] is the simplest way to provide causality, however, this approach does not offer concurrency as it blindly marks all operations as dependent, having a significant impact on performance. 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 (optimizations exist to avoid storing all the previous executed operations). When an operation is propagated across replicas, the causal history is appended to the message, 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 casual history of A, then all operations contained in the causal history of B will also be included in 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 [**gentlerain**] 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 [**cure**] 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 [**saturn**] 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 relevant. 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 when compared to the Cloud, 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[**cops**], 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. At the global level, there are two options, either each data centre contains a complete copy of the data or it contains only a subset, in which case the data each data centre will store must be determined. Inside data centres, there are also two main options to determine the data stored at each server. Random placement can be used to achieve load balancing or usage patterns can be analysed to infer the data relations and make an informed decision. Usage patterns can also be used by geographically distributed systems to store data closer to the user location when data centres do not contain full copies of the data as it can result in high costs. Manually deciding where each data item would be stored is not feasible, making the case for an automated system that can

indicate the ideal locations. Another challenge is that usage patterns can change considerably with time, due to new features introduced or because users simply changed locations or interests. An ideal system would detect these changes and react autonomously.

2.5.1 Data Placement in Data Centres

Typically the only focus inside data centres is on partitioning the data per server in a fair way between all servers according to their resources. In order to achieve this, most systems, such as Cassandra [**cassandra**] and Dynamo [**dynamo**], resort to a scheme based on consistent hashing to partition the data. 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. Possible performance gains from storing items frequently accessed together in a smaller number of nodes have been overlooked by these systems. The schemes proposed by the authors in [**yu2015location**, **yu2017framework**] address this problem by focusing on increasing co-locality while still ensuring load balancing. Fine-grained control is used over the data relations of each data item to determine where it should be stored.

2.5.2 Geographically Distributed Data Placement

The recent growth of geographically distributed public Clouds encouraged the appearance of globally available applications. These applications are distributed across two or more data centres in different geographic regions and use a considerable amount of servers, a system composed of thousands of servers are not unusual.

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 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 complication 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.

Volley [**volley**] 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 [**yu2015location**, **yu2016sketch**, **yu2017framework**], where the data placement problem is modelled into a hypergraph partitioning problem. The hypergraph represents the data relations and the placement locations are extracted by partitioning the hypergraph. The work presented in [**yu2015location**] 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 its focus is on maximizing co-locality, which improves the performance when read requests include multiple keys [**multigethole**]. 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 [**yu2016sketch**] the authors argue that the overhead of the centralized design presented in their previous work [**yu2015location**] 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 [**yu2016sketch**, **yu2015location**, **yu2017framework**] 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 [**atrey2018scalable**], 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.

The work presented in [**huang2019yugong**] 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 [**zhou2018optimizing**] 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 on information provided by the application to identify cl ient 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.

2.5.3 Data placement for Edge Computing

The presented works have studied the data placement problem for geographically distributed data centres on the 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, move data closer to where it is required to improve overall performance, which is still valid when considering Edge Computing. However, these works focus on a global choice over where data is placed, while our system requires a more fine-grained control, we are interested on more local choices as our goal is to have multiple edge devices running replicas with very limited resources extending a data centre.

Data will move around the edge as required, possibly not staying in the same replica for a long period of time. Hence we need a scheme that supports these frequent changes capable of quickly and autonomously adapting, so a highly centralized and computational intensive schemes [**volley**, **yu2015location**, **zhou2018optimizing**] which take a considerable amount of time to complete are not applicable.

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. This was not a problem as most applications did not need such strong guarantees to begin with, and the ones that did could implement them over this model where required. 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 Sub 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 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.[**dynamo**]

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, 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.

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.

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.

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[**scutllebutt**], 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[**hayashibara2004spl**] 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 the node is marked as 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[**cops**] 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[**chainreaction**] 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 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 writer 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 to Kth replica 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 [**chainreaction**]. It requires only one round most of the time, unlike COPS where two rounds are required every time values are updated during the transaction.

Chapter 3

Proposed Work

In this chapter, we will start by introducing in Section 3.1 the system model in which our solution will be based upon. In Section 3.2 we will present a solution sketch and break it down into the two main challenges that must be addressed. In Section 3.3 we define how the evaluation of our implementation will be done. Lastly in Section 3.4 we define the phases into which our work will be divided, considering the previously identified challenges, the evaluation phase and the writing phase.

3.1 System Model

In this section we present the system model for our replication system that extends the storage model from the cloud onto to the Edge. We consider two types of replica, differentiated by their behaviour and lifetime: stable replicas and dynamic replicas. Both types of replicas offer the same functionalities, not to be compared with cache systems, such as CDN where operations are read-only.

- **Stable Replicas:** These replicas are defined by having a stable dataset for which they are responsible to replicate, possibly changing only when a replica joins or leaves the system. They are located inside data centres and have a considerable amount of resources available and uptime.
- **Dynamic Replicas:** Opposite to stable replicas, the replicated dataset might change often, they adapt according to the usage patterns and the available resources. These replicas run on the edge, which is composed of various types of devices that can have very different characteristics. These devices are also more prone to failures due to the instability of the environment when compared to data centres, in our initial solution we consider failures can last for a long period but eventually recover.

The available resources vary greatly per replica (i.e., device) due to the high heterogeneity variety of devices on the edge. As each device has a primary function it must fulfil, only the remaining resources are free to be used by secondary services. Initially we will assume that the resources available per device are static for our initial solution. In our initial solution we will also assume a static membership where no Edge devices are added or removed. We will then work on a more realistic solution where the resources available in the edge devices might change during their lifetime due to peaks where more resources to fulfil its primary function are required, hence reducing the resources available for secondary services and need to adapt correspondingly. In this solution a dynamic membership will also be considered where edge devices can leave and join freely. To achieve this we assume that there is a service providing us with the required membership information as well as metadata about the device itself, such as the IP, location of the device, and its available resources.

3.2 Solution Sketch

In this section, we present a sketch of our solution and the required steps, adjustments will most likely need to be made during the development process. In some cases, we will mention how various challenges are addressed in an initial solution that will be developed first and ensures the basic functionality of the system, as well as a more complex solution with some optimizations. However, due to the time constraints, development of the complex solution is dependant on how the work progresses.

Our goal is to create a distributed data store system that can simultaneously use the Cloud and the Edge to take the best of both models, such data store should support genuine partial replication, at least on the replicas running on the edge. It should also provide causal+ consistency. We assume the data centre has the complete set of the data while edge replicas will only have a small subset of the data.

We will extend the storage model from the cloud onto the Edge, by creating replicas closer to the end-user where a limited amount of data will be stored and light might be computations performed.

3.2.1 Data Placement

As edge replicas have limited resources available, we will need to determine which data to store and replicate at each edge replica. An autonomous data placement scheme is the most adequate due to the edge being a very dynamic environment, it needs to quickly react to access patterns changes and available resources.

Such a scheme has two main challenges, first, it must decide which data is relevant to each user, secondly, it has to decide where this data should be replicated, finding a balance between performance gain and synchronization costs. To address the first challenge we must determine data to user relations. An initial solution will analyse the request logs to

determine the previously mentioned relations, in case it is the user first access it is not yet possible to determine these relations hence no action can is taken. A more complex solution could receive application provided information, like *friendship relations* in the example of Facebook or *follows* in the case of Twitter.

The second challenge is addressed in our initial solution by copying the data to the a close replica when the user connects to the system, based on the user IP or its location when available. A more complex solution could try to prefetch the data to the edge considering the previous locations of the user. Sometimes data might have to be removed from the edge replicas to free up space for other users or to free up space as commanded by the edge device. One efficient way to achieve this could be to use an eviction policy such as LRU, but modified to synchronize unconfirmed operations with a data centre or other replicas to ensure durability.

3.2.2 Replication Protocol

As data can be changed in any replica, a protocol is required to connect the two layers to handle the replication and synchronization.

Such a protocol must be highly dynamic as edge replicas will be changing the dataset they replicate very often, and for the solution with dynamic membership the edge replicas can leave and join at any time. Our solution needs to integrate the replicas on the edge with the replicas inside a data centre, as data can be modified in both layers. Ensuring durability of update operations executed at the edge efficiently is one of the main challenges for such a scheme since propagating operations to the data centre before returning to the user would defeat most of the purpose of having replicas at the edge in the first place. We see two possible solutions :

- Replicate the operation to a certain number of close replicas at the edge before returning to the user. Upon a failure the user would connect to one of the other replicas. This would ensure fault tolerance up to the number of replicas minus one.
- Replicas could return immediately and would gradually mark operations as permanent by adding metadata to the responses. The client library would keep track of unconfirmed operations by analysing the metadata. Upon a failure it could connect to a different replica or directly to the data centre, check if the unconfirmed operations were propagated or not, replaying the ones that were not.

As each data is dynamically added or removed from edge replicas, the replication scheme must efficiently support these changes, as they will be constantly happening.

3.3 Evaluation

To evaluate our solution we will focus on measuring the user-perceived latency, the rate of requests satisfied at the edge (co-locality), the throughput and the overhead of our

implementation.

To evaluate our solution we will build or adapt an existing application to use our storage layer. One application type we think could make the most of our solution is location based augmented reality games, such as Ingress[1] or Pokemon Go[2], where users interact with other nearby users or objects in their location.

The evaluation will be conducted using a data centre to run the stable replicas, the university cluster to simulate the replicas on the edge as well as the clients. The workloads will be generated using a generator, such as the Yahoo Cloud Serving Benchmark [**cooper2010benchmarking**] which is highly customizable. The testing workload will need a notion of workload to simulate the type of user interaction seen in location based applications.

3.4 Work Plan

In this section we present our work plan, defining the main tasks and respective subtasks. We will have one main development task that will consist of 3 phases. We will also try to submit an article covering the dynamic replication protocol developed for our system.

The tasks schedule can be seen in figure 3.1 and are divided as follows:

- Task 1 :
  - Task 1.1 : Data Placement Scheme Development
    - Task 1.1.1 : Design
    - Task 1.1.2 : Implementation
    - Task 1.1.3 : Testing and Optimizations
  - Task 1.2 : Replication Protocol Development
    - Task 1.2.1 : Design
    - Task 1.2.2 : Implementation
    - Task 1.2.3 : Testing and Optimizations
  - Task 1.3 Develop final system with integrated components
    - Task 1.3.1 : Design
    - Task 1.3.2 : Implementation
    - Task 1.3.3 : Testing and Optimizations
- Task 2 : Evaluation
- Task 3 : Writing the Dissertation

[1] https://www.ingress.com/
[2] https://www.pokemongo.com/en-us/

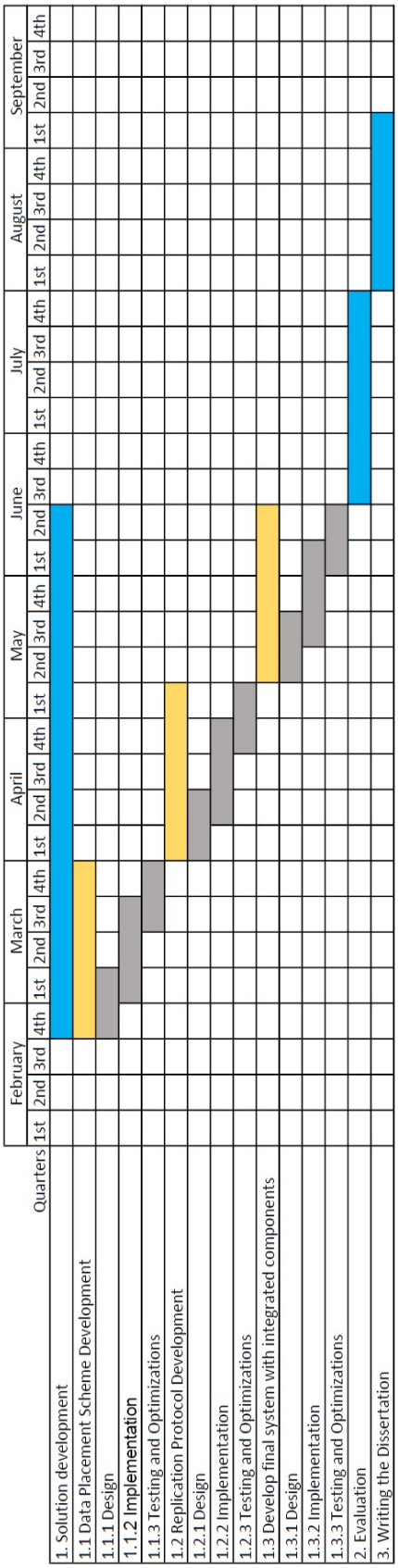

Figure 3.1: Work plan