NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

DIOGO ANTÓNIO ROCHA ALMEIDA

BSc in Computer Science

BYZANTINE FAULT TOLERANCE IN LARGE SCALE EDGE REPLICATED SYSTEMS

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

BYZANTINE FAULT TOLERANCE IN LARGE SCALE EDGE REPLICATED SYSTEMS

DIOGO ANTÓNIO ROCHA ALMEIDA

BSc in Computer Science

Adviser: João Carlos Antunes Leitão
Associate Professor, NOVA University Lisbon

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2024

Abstract

Storage systems play an essential role in supporting cloud applications by providing scalable, reliable, and efficient data management solutions. Current cloud infrastructures, although capable, fall short of meeting the stringent latency requirements of novel and highly client centric applications. The advent of edge computing aims to bridge this gap by bringing computations, and hence data manipulated by them, closer to the end users. However, the susceptibility of edge machines to malicious activities poses a significant obstacle. Unlike data centers, edge nodes lack a robust security perimeter, making them vulnerable to be compromised, which can impact the integrity and functionality of applications residing at the edge.

In this work, we plan to address the previous challenges by designing and developing an edge-driven storage system resilient to byzantine faults, addressing the security concerns prevalent in the edge environment. Most existing works in this domain either neglect the unique challenges of edge computing, fail to consider byzantine security issues, or lack comprehensive replication and consistency guarantees. We propose a system that transparently deploys to the edge, supports arbitrary applications while ensuring consistency and persistence guarantees, and mitigates the impact of malicious edge nodes on data integrity and confidentiality. To this end, we will design a data and consistency model that operates under a defined byzantine threat model. Key challenges include expanding data storage to edge clients, partially replicating data across edge nodes, and minimizing the impact of malicious nodes with minimal overhead. To evaluate our proposal we plan to compare it with state-of-the-art storage systems used in industry that are deployed in data centers, other edge-driven storage systems, and other systems that provide some security guarantees.

Keywords: distributed storage systems, decentralized systems, edge computing, byzantine fault tolerance, causal+ consistency, partial replication

Resumo

Os sistemas de armazenamento desempenham um papel essencial no apoio às aplicações *cloud*, fornecendo soluções de gestão de dados escaláveis, fiáveis e eficientes. As atuais infraestruturas de computação *cloud*, embora capazes, não conseguem satisfazer os rigorosos requisitos de latência das novas aplicações, que são altamente focadas no cliente. O surgimento da computação *edge* visa superar esta barreira, aproximando os cálculos e, consequentemente, os dados por eles manipulados, dos utilizadores finais. No entanto, a suscetibilidade das máquinas na *edge* a atividades maliciosas representa um obstáculo significativo. Ao contrário dos centros de dados, estas máquinas não dispõem de um perímetro de segurança robusto, o que os torna vulneráveis a ficarem comprometidos, podendo afetar a integridade e a funcionalidade das aplicações que residem na *edge*.

Neste trabalho, planeamos abordar os desafios anteriores, concebendo e desenvolvendo um sistema de armazenamento orientado para a *edge* e resiliente a falhas bizantinas, abordando as preocupações de segurança prevalecentes no ambiente *edge*. A maioria dos trabalhos existentes neste domínio negligencia os desafios únicos da computação *edge*, não considera as questões de segurança bizantinas ou não oferece garantias abrangentes de replicação e coerência. Propomos um sistema que se instala de forma transparente na *edge*, suporta aplicações arbitrárias, assegurando simultaneamente garantias de coerência e persistência, e mitiga o impacto de dispositivos na *edge* maliciosos na integridade e confidencialidade dos dados. Para o efeito, conceberemos um modelo de dados e de coerência que funciona sob um modelo de ameaça bizantino definido. Os principais desafios incluem a expansão do armazenamento de dados para clientes *edge*, a replicação parcial de dados entre máquinas *edge* e a minimização do impacto de máquinas maliciosas com o mínimo de sobrecarga. Para avaliar a nossa proposta, planeamos compará-la com os sistemas de armazenamento de última geração utilizados na indústria que são implementados em centros de dados, outros sistemas de armazenamento na *edge* e outros sistemas que fornecem algumas garantias de segurança.

Palavras-chave: sistemas de armazenamento distribuídos, sistemas descentralizados, computação *edge*, tolerância a falhas bizantinas, coerência causal+, replicação parcial

Contents

List of Figures

1 Introduction

Storage systems play a crucial role in supporting and optimizing cloud applications by providing scalable, reliable, and efficient data management solutions. These systems offer horizontal scalability, enabling users to adjust storage capacity based on needs [13, 49, 35]; redundancy mechanisms commonly based on replication to mitigate the impact of hardware failures [13, 35, 61]; and efficient APIs and protocols that allow specific data to be retrieved and modified with certain consistency guarantees [49, 57, 35]. More advanced systems include security mechanisms to encrypt data in storage and provide access controls, data lifecycle management to archive unused data, and disaster recovery backups [19, 49, 25].

Currently, having applications fully deployed in cloud infrastructures, inside data centers that are distant from users, has become insufficient to effectively support novel user-centric applications having stricter requirements on operation latency to work properly [60, 42]. Edge computing addresses this and other limitations of cloud computing by bringing data and computation closer to end users through computational resources that reside between the cloud and end users.

A problem with edge computing is the significantly increased susceptibility of machines[1] at the edge to become malicious or enter the system while already being malicious [66]. The compromise of these machines also compromises the functionality and integrity of the resident applications, limiting what can be done at the edge. These concerns are not as prevalent in cloud computing because data centers are enclosed withing a strong secure perimeter [52].

In this work, we plan to design and develop an edge-driven storage system that is resilient to byzantine (arbitrary) faults. To the best of our knowledge, most of the work done in this area does not consider the edge environment, does not consider byzantine security issues, or failed to provide useful replication and consistency guarantees for applications built on top of these solutions.

[1] In this document we use the words machines, processes, and nodes interchangeably.

1.1 Objective

In this dissertation, we plan to address the operation of edge computing under the byzantine fault model by designing and developing a storage system that is resilient to byzantine faults and that takes advantage of the edge landscape. More specifically, a storage system that can transparently be deployed to the edge supports an arbitrary application, providing clearly defined consistency and persistence guarantees while being resilient to potential malicious edge nodes that can compromise the integrity and confidentiality of the sensitive data of the application.

To achieve this goal, several challenges must be addressed. The system needs to expand the data storage to clients while providing some guarantees of data consistency, meaning that the data should be presented to and modified by clients on some logical time/event order. Because data is being expanded to machines at the (far) edge of the system, it is clear that no single machine should hold all the data, and it must be partially distributed in a way that considers the heterogeneous capabilities of the overall system. Additionally, the effect of malicious nodes must be mitigated, and the mechanisms employed should present minimal overhead on the running system while upholding the aforementioned properties. This also includes the protocols for membership and communication (overlay network) that guarantees that communication between machines and the rest of the system is not disrupted.

1.2 Contributions

The expected contributions of this work are as follows.

1. Design and implementation of a data model with properties that enables and eases the implementation of the replication and security mechanisms of an edge storage system, while providing useful data retrieval and modification operations.

2. Extension of the design and implementation of an existing overlay network that provides resilience from malicious nodes that disrupt the communication of other nodes in the system to better adapt to the edge environment.

3. Design and implementation of a distributed storage system that expands to to edge nodes leveraging the developed data model, providing data consistency and persistence guarantees through a replication protocol, and security guarantees on an environment where a number of nodes can be malicious and do arbitrary actions to disrupt the system.

4. Experimental comparison of our proposal with state of the art systems. These systems can range from popular ones used in the industry (usually only employed across data centers) to edge-driven systems and systems that provide security guarantees on more decentralized settings.

1.3 Research Context

This dissertation is conducted in the context of a research project led by NOVA School of Science and Technology and funded by the European Commission: the TaRDIS (Trustworthy And Resilient Decentralised Intelligence For Edge Systems) European project [62].

1.4 Document Structure

The remainder of this document is organized as follows.

- Chapter 2 presents the relevant concepts that are the basis for the elaboration of the work in the dissertation. More specifically, it introduces edge computing, existing storage systems, and the concepts of consistency models, data models, and byzantine model. These concepts are useful for understanding the various options and their advantages and disadvantages in the context of the problem addressed in this work. Combining these components yields the basis for a proposed novel solution for an edge-driven storage system that operates with potentially malicious nodes.
- Chapter 3 details the future work to be conducted, including the goals and assumptions for the target system, the challenges involved, the initial strategy to overcome these challenges and achieve the goals and the proposed scheduling for this work.

2

Related Work

In this chapter, we explain the relevant concepts that form the basis for the elaboration of this dissertation, while presenting and analyzing relevant related work. The chapter is structured as follows: in Section 2.1 we introduce distributions models such as cloud and edge computing; in Section 2.2 we discuss some distributed abstractions that are not directly tackled but are relevant for this work; in Section 2.3 we introduce the concept of distributed storage systems and how they are constituted; in Section 2.5 we introduce the concept of data replication, supporting replication and consistency models and replication protocols; in Section 2.4 we introduce data models; in Section 2.6 we introduce the problem of byzantine fault tolerance, and present some existing mechanisms and protocols that provide byzantine fault tolerance; in Section 2.7 we present existing solutions that use some of the previously covered concepts; finally, in Section 2.8 we summarize this chapter.

2.1 Distribution Models

Distribution models are paradigms that distribute computing resources across multiple machines which can potentially be located at different geographical locations. These models were designed to address some of the limitations of traditional computing models, such as the inability to scale resources efficiently and the inability to operate when there are machine or network outages. All these models have an architecture composed of multiple machines that coordinate to achieve a certain goal.

2.1.1 Cloud Computing

Cloud computing [9] is a distribution model in which IT services are provided through massive computing units over the Internet, typically housed in data centers. This paradigm shift allows users to access computing resources, such as processing power, storage, and networking on demand, eliminating the need for users to acquire and maintain their own computing infrastructure, which is costly and inefficient. Cloud computing users can deploy applications in the cloud, with the cloud provider handling operational aspects.

Despite these advantages, cloud computing has several drawbacks. It follows a centralized architecture, making it susceptible to data center outages that can render applications inoperable. Cloud computing is also not suitable for applications that have stringent latency requirements, as data centers are usually located far from the users. Security concerns arise because users must trust cloud providers to adhere to protocols and protect data from destruction, modification, or unauthorized access. Scaling resources in cloud data centers presents difficulties, particularly in efficiently handling the surge in data traffic between servers, storage, and other components, which can lead to bottlenecks and latency issues.

Geo-replication has emerged as a solution to some of these challenges by extending cloud computing to multiple data centers in different geographical locations. This technique mitigates latency issues by deploying applications closer to users through multiple data centers, ensuring continued operation during data center outages, and facilitating resource scaling across diverse locations. However, it introduces new challenges, such as the need for efficient cross-data center replication, considering the differences in machine proximity and network infrastructure robustness within and between data centers.

2.1.2 Decentralized Systems

In decentralized systems, no single entity controls the system. Instead, the system is composed of multiple independent entities that coordinate with one another. Because there is no single point of coordination, these systems have the advantage of being robust, because the failure of a portion of the system does not completely disrupt the entire system, which can continue to operate normally. Regarding independence, no single entity has control over all the data and can dictate how the system should operate, which is also a strong motivation for adopting this architecture.

Compared to a classic client-server architecture, it is much more complex to coordinate all the resources of the system while maintaining the required correctness guarantees and performance expectations. Heterogeneity is common because individual nodes are more prone to differ from each other in their amount of resources. Nodes are more susceptible to attacks, as they have a weaker network perimeter that is easier to penetrate and compromise, and to have deviant behavior, as they may have self beneficial or even purely destructive reasons to be deviant and exploit the system. Unlike data centers, machines are usually not close to each other and can be located at different geographical locations. Moreover, nodes may be unavailable indefinitely, and their data may be permanently lost, which must be considered when designing the system.

2.1.3 Edge Computing

Edge computing is a distributed model that brings computation and data storage closer to the source of data generation, the end users, rather than relying solely on a centralized cloud infrastructure [60]. It combines the advantages of cloud computing and decentralized

systems while having some of their drawbacks that need to be addressed. Some tasks and data are still processed in the cloud, whereas others can be fully handled at the edge. The edge coordinates with its servers, devices, and the cloud to handle user operations quickly at the edge, providing a seamless experience for the end user.

This approach is designed to address the limitations and challenges of traditional cloud computing, particularly in scenarios where low latency and real-time processing are critical. Latency is reduced because operations do not have to travel far to reach the cloud for a user to obtain a reply. Availability is improved because of the robust decentralized nature of the edge and because the edge can continue to partially operate, even when there is a cloud outage. Regarding the scalability of resources, the edge is composed of machines with their own resources, which can be scaled independently of the cloud. This alleviates the problem of scaling the network infrastructure of the cloud because the edge can handle most of the load.

2.1.3.1 Edge Architecture

The architecture of the edge is hierarchical [42], where the cloud resides at the top edge devices, which are the endpoint clients where data is generated and consumed reside at the bottom, and edge servers with varied capabilities reside at the middle.

Edge resources are categorized by the following properties: capacity, which includes processing power, storage capacity, and connectivity (which includes security perimeters); availability, which is the probability that the resource is reachable; and domain, which is the operational support the resource has for an application, meaning that it either completely supports the application operations (application domain) or only supports the activities of a given user (user domain).

Closer to the top, edge resources have the highest capacity and availability, and are in lower quantities. Regarding availability, it is usually assumed that the cloud may face temporary outages but does not permanently lose data, unlike edge servers, which may. Going down, the resource capacity and availability tend to decrease, and the quantity of these resources tends to increase. Application domain is usually supported for resources owned by larger entities (e.g., companies and governments) and user domain for personal owned resources, such as laptops and smartphones.

The challenges of edge computing [42] include the efficient dynamic and partial replication of application data, following a consistency model; allocation of resources considering load balancing, task offloading (where certain tasks should be processed), scalability, and fault tolerance; enabling computations or clients to move when machines suffer outages or data is required elsewhere; security and data protection for applications, including data integrity, access control, and data confidentiality; and lightweight and decentralized monitoring of resources for telemetry, statistics, or from a fault tolerance perspective to detect faulty nodes. This dissertation focuses on the first three challenges.

2.2 Distributed Abstractions

In this section, we introduce distributed abstractions that are relevant to the context of the concepts discuss in this work.

2.2.1 System Models

Implementing distributed algorithms requires defining some underlying system model. This model defines assumptions about the system components, their environment, and time to facilitate the reasoning of the designed algorithms.

Regarding time assumptions, in the synchronous model, there are bounds on message delays between processes and process speeds, making it easier to reason regarding the phases of the protocols. By contrast, in the asynchronous model, there are no bounds. This model is analogous to how the real world works, although it is much more difficult to implement. Most systems use a semi-synchronous model to simplify reasoning, where there are bounds on message delays but not on process speeds.

Real world networks, as a result of being asynchronous, are also unreliable, implying that messages may be lost, duplicated, mutated, or delivered out of order. Therefore, the system must handle these properties by defining a protocol. A perfect link is a protocol that ensures that messages from a sender to a receiver are delivered eventually, without duplication, and in the order in which they are sent, implementing a reliable communication channel. Fortunately, it is fair to assume that reliable network transport protocols such as TCP [56] or QUIC [39] provide the implementation of a perfect link.

Assumptions must also be made regarding the failure of the system components. In the crash-stop model, the processes fail by stopping and do not recover. In contrast, in the crash-recovery model, processes may recover later and rejoin the system with the possibility of maintaining a previous state that was stored in persistent storage. There is also the byzantine fault model, in which processes may fail by deviating from the protocol and performing arbitrary actions. In Section 2.6, the algorithms used in this model are discussed.

2.2.2 Membership Abstractions

Membership refers to the perception of the system's nodes to other nodes. Each node maintains information about other nodes in the system, such as their addresses and other metadata, which can be used to communicate with them. In a system with full membership, each node contains information about all the other nodes. In systems with a large number of nodes, full membership is not scalable. With partial membership, each node only has information regarding a subset of nodes, its neighbors.

Overlay networks are virtual network layers on top of a physical network infrastructure that provide a partial membership protocol. These protocols manage the membership of each node and react to events such as nodes joining and leaving the system (churn).

Overlays can be either structured or unstructured. Structured overlays organize nodes in a structured manner such as trees or rings. In unstructured overlays, there is no explicit organization of nodes, and they are connected to each other in a random manner. Partially structured overlays contain elements of both structured and unstructured overlays. They typically maintain a certain level of structure while exhibiting a degree of flexibility in node interactions.

HyParView [41] is an unstructured overlay that is robust to churn and provides eventual full connectivity between nodes (the membership graph is fully connected). This is often a desirable property that allows a message to reach any node in the system from any other node. Kademlia [47] is a tree-shaped structured overlay that provides a distributed hash table (DHT) abstraction that provides efficient lookup of values given a key, and of other nodes in the system. PlumTree [40] and Bias Layered Tree [11] are partially structured overlays that provide tree-shaped structures, with the latter being more adjusted to operate at the edge.

2.2.3 Communication Paradigms

Communication paradigms offer a means for nodes to communicate with each other and propagate (or disseminate) information throughout the system. Various paradigms can be used to provide different properties and tradeoffs. Some paradigms are used to build other distributed abstractions.

2.2.3.1 Broadcast & Multicast

Broadcast is a paradigm in which a node sends a message to all other nodes in the system, even though it may not be directly connected to them. Multicast is a similar paradigm, in which a node sends a message to a selected subset of nodes in the system.

There are stricter versions of broadcast and multicast models in which the order of delivery of messages by nodes is restricted. For example, in FIFO broadcast/multicast, messages are delivered in the order in which they are sent by the sender, and in totally ordered broadcast/multicast, all broadcast messages from every node are delivered in the same order by all nodes.

2.2.3.2 Gossip

Gossip (or epidemic broadcast) [14] algorithms provide a means of disseminating information across nodes in the system. They involve nodes sending messages to other nodes, which, in turn, send messages to other nodes until all nodes receive the message. Gossip algorithms are used in overlay networks to implement higher-level mechanisms such as broadcasts or lookups.

There are various types of gossip that provide different properties related to the speed of dissemination, network traffic, and network (bandwidth) cost. On a point-to-point level,

eager push gossip involves spontaneously sending messages as soon as possible, which increases propagation speed. However, if messages are large, it incurs a higher network cost. In contrast, in pull and lazy push gossip, messages are sent only when one of the nodes requests the message from the other.

There are also many gossip types at the group level. For example, flood gossip propagates messages with eager push where nodes, upon receiving a message for the first time, send it to all its neighbors except the sender. This gossip type is the fastest in disseminating messages but incurs the highest network cost as nodes receive many repeated, redundant messages. Anti-entropy executes periodic message pulls from a neighbor at a time. Random walks are messages that "walk" through the overlay (are sent to a random neighbor recursively) until they reach a certain number of nodes.

2.2.3.3 Publish-Subscribe

Publish-subscribe [53] (or shortened to pub-sub) is a useful, higher level, paradigm where, similar to multicast, messages are disseminated to a subset of nodes. However, messages contain a tag (or topic) to which certain nodes subscribe, and messages are only delivered to those nodes. Unlike multicast, the sender does not necessarily need to know who is subscribed and who will deliver the message.

The implementation of a pub-sub protocol may depend on the underlying overlay network used. DHT overlays usually allow for more efficient pub-sub protocols because they enable efficient lookups for nodes with specific information, such as subscribed topics. In unstructured overlays, pub-sub protocols may require broadcasting methods to ensure that all the subscribed nodes deliver the message.

2.2.4 Consensus

The consensus (or agreement) problem [37] involves achieving agreement among a set of processes for a certain value. Each process has an initial value that it proposes, and all correct (non-failing) nodes eventually decide, once, on the same value, that one of the processes proposed. This is a useful problem to solve as it can be used to implement other abstractions such as state machine replicas, which we discuss in Section 2.5.2.

FLP [21] proves that in a system in which processes can fail by crashing and messages can be delayed in the network indefinitely (similar to real world networks), it is impossible to solve the consensus problem deterministically. This impossibility result is due to the fact that, under these condition, there is no way to distinguish between a process that has failed and a process that is taking a long time to respond. Therefore, some properties must be relaxed to solve a weaker but still useful version of the consensus problem.

Paxos [36] is a consensus protocol that solves a relaxed version of the consensus problem where correct processes are not guaranteed to eventually decide a value (termination property). Multi-Paxos [63] is an optimization of Paxos, in which only one (leader) process proposes values. This allows the decision to be made in fewer communication steps.

2.3 Distributed Storage Systems

Storage systems support applications by efficiently storing, retrieving, and managing the data. There are various types of these systems, tailored to specific application needs. These systems store application data and offer a set of operations to retrieve and modify data with the desired structure. Data is stored directly in persistent storage devices connected to the machine running the system, such as hard or solid-state drives.

Distributed storage systems are used to efficiently scale applications that handle large amounts of data with extensive access. These storage systems are executed by leveraging the resources of the multiple interconnected machines. Distribution allows the system to scale out, adding more resources through more independent machines and devices, which is cheaper, provides better fault tolerance, and can be done without disrupting the currently running system, compared to scaling up, upgrading the hardware of existing machines. Consequently, distributed algorithms must be employed such that these machines can coordinate data between themselves.

The data model of the storage system structures how the data is stored and retrieved, and defines and exposes the operations that can be performed on the data. We further analyze data models in Section 2.4.

The system's distribution module manages the manner in which data is stored throughout the machines. It depends on how often data is accessed, how efficiently operations are performed, and how persistent the data needs to be. Replication techniques are used to provide data persistence by having copies of data, called replicas, in different machines. However, the consistency of application data in replicas must be upheld. Consistency can be broken when different replicas receive conflicting operations in different orders. We delve deeper into the issues of replication and data consistency in Section 2.5.

Clients interact with the application that holds the semantic functionality of the service. Applications, in turn, interact with the storage system through the data model's interface to obtain or write information. The data model of the storage system interacts with its distribution module, which coordinates data and operations with other nodes. The data is structurally stored and retrieved from the storage device using the data model.

A storage system is deployed in a distribution model. As discussed previously, different distribution models exhibit different properties. As such, storage systems can be specifically designed to operate in certain distribution models, using their underlying properties to delineate the system's design and using some assumptions on the model to simplify the system and potentially improve its performance.

Data center storage systems operate in centralized, clustered environments, where machines are close to each other. These are designed to perform efficient data replication within the data center. They can also be extended to geo-replicated setups, where the main challenge is to efficiently replicate data between multiple geographically separate data centers.

Decentralized storage systems operate across non-clustered machines. The main

challenges include replicating data in a robust way and regarding its locality and heterogeneous capabilities. Edge-driven storage systems have similar properties and challenges as decentralized storage systems, but they can occasionally rely on the cloud to implement mechanisms and provide guarantees. Deriving the hierarchical nature of the edge architecture, solutions [3, 22] tend to employ a tree-like structure node relationship, where the root is the cloud and the node children are edge servers.

2.4 Data Models

Data models are abstract representations of the structure of data and their relationships within the database. These models provide a way to organize, store, retrieve, and manage data efficiently. Data models define application programming interfaces (APIs), which are operations that can be performed on data, and the semantics of these operations (e.g., `get(key)` and `put(key, value)`). Different data models are designed to have different characteristics and purposes, and applications choose storage systems that use data models that best fit their requirements as the most important factor.

With the expansion of the Internet and its services, the demand for scalable databases has increased. Traditional relational databases have struggled to meet the performance requirements for this usage growth. In response to this, NoSQL (Not Only SQL) databases emerged as a solution to the limitations of relational databases. Diverse NoSQL data models have been designed to provide a more flexible way to organize data, thus enabling simpler and more performant databases. NoSQL data models have become popular in distributed storage systems, where scalability and performance are important.

In NoSQL data models, the basic operations are defined as create, read, update, and delete (CRUD). More complex operations can be defined by using these basic operations. Aggregation operations produce results based on the selection of a subset of data. Data models are designed to operate best with a chosen set of operations, or with a trade-off between them.

How data is structured according to the data model also affects how data can be partitioned. Data is partitioned into multiple subsets to optimize operations that require only a subset of the data. In distributed environments, data is also partitioned to scale the storage of data across multiple machines, to place certain data closer to where its access is more common, reduce latency, and increase the concurrency of operations over the system. By partitioning, there must also exist a mechanism to localize the machine(s) in which the target data resides. However, the operations that require data from multiple partitions are slower and more complex.

2.4.1 Key-Value

The key-value is the simplest data model that represents and organizes data as a collection of key-value pairs. Each piece of data is stored as a pair of values containing a unique

identifier, key, and the associated data value. The key is used to retrieve the data value. This data model is designed to be simple and efficient for handling the CRUD operations. Finding the key usually requires constant computational effort. These stores are also designed to be easily scalable as data can be partitioned per data pair because there is usually no relationship between the data pairs. This data model serves as the basis for more complex data models.

The drawback is that it is not efficient for operations that require aggregation of data, as the data is not structured in a way that allows it to be easily aggregated. It also falls short of providing efficient indexing, as the only possible index is key. The simplicity of this data model makes it more suitable for simpler applications that do not require complex operations, relationships between data values, or data aggregation such as caching or session stores.

2.4.2 Document Based

The document based data model is designed to store data in an intuitive and flexible manner. Similar to the key-value data model, this model attributes a unique key to each document, which in this case, is the data value. Unlike the basic values in the key-value model, documents are self-contained pieces of data containing semantic information regarding certain entities in the application. Documents are usually stored in a format that is easy to parse, such as JSON (JavaScript Object Notation) or XML (Extensible Markup Language).

Generally, these documents do not have a fixed structure, meaning that different documents can have different fields, and that the same field can have different types of data. They can also be nested, meaning that a document can contain other documents, thus allowing more complex data structures. Documents are organized into collections, which are groups of documents with similar structures. Secondary indices can be created to accelerate searches, ordering, and other aggregation operations based on the data field values. This data model allows the database to scale horizontally as data can be partitioned per document.

2.4.3 Column Oriented

Column oriented data models are designed to store data in column tables. Unlike the relational data model, there is no association between the tables and columns. Data is stored by column, separated from other columns, and the data in each column is of the same type. Each column has a unique name and each row has a unique identifier. Rows in different columns may be identified using the same identifier to associate their values with a data entity. Generally, each column may also be grouped into column families, which are groups of columns in which the data is expected to be accessed together.

In this data model, data is usually partitioned per column family such that data records with relationships between them are grouped together. This data model is designed to be

efficient for data aggregation because data is structured in such a way that all the data to be aggregated reside closely together. In the case of distributed systems, data may reside in the same machine or in the same cluster of machines. It allows for the efficient compression of data, as the data in each column is of the same type. Typical applications that use this data model require analytical processing such as data warehousing.

2.4.4 Conflict-Free Replicated Data Types

Conflict-free Replicated Data Types (CRDTs) [59] are a class of data types designed for distributed systems in which multiple replicas of data exist and updates to the data can occur concurrently. They ensure that all replicas converge to the same state by resolving conflicts between concurrent updates in a deterministic manner instead of using a centralized service or other distributed coordination mechanisms. We further address the problem of conflict resolution in Section 2.5.3.1.

Conflict resolution solutions are either designed for specific data types or specific applications, which makes them difficult to generalize, are not sufficiently flexible, such as a policy where a certain type of operation has priority over others, or are not sufficiently efficient, such as distributed mechanisms (e.g., consensus to decide which operation is performed). CRDTs are designed to be general, applicable to any data type, simple to implement, and efficient in terms of space and time complexity.

There are two types of CRDTs: state-based and operation-based. In summary, in state-based CRDTs, a new state is created when an update is performed, as is typical. This new state is then propagated to the other replicas and merged with their current state according to CRDT's definition. This state can be either the full state or an independent object of the data. In operation-based CRDTs, updates are propagated to replicas and applied to the state when the precondition of update, a function of state and update, is satisfied. It has also been proven that state-based CRDTs are equivalent to operation-based ones [59].

In both types, merges are commutative, implying that the order in which they are performed does not affect the final state when all operations are performed. There may be applications where this property cannot be upheld to its functionality, or where the order in which the operations reach the system is important. Therefore, CRDTs are not suitable. In such cases, other distributed coordination mechanisms must be used.

A useful example of a CRDT application [59] is the set data structure. A set is a data structure that stores a collection of unique elements, which is useful in data models. A state-based CRDT implementation of an add-only set is the union of propagated and current replica sets. A normal set can then be implemented with two add-only sets, A and R, one for added elements and the other for removed elements. Querying the set returns $A \setminus R$.

These data types are useful for implementing data models of distributed storage systems because they provide a way to resolve conflicts efficiently by simply using the

definition of the data type. There is a panoply of distributed storage systems that use CRDTs in their data models, from which the user can choose, depending on the application requirements. Some systems provide users with the ability to define custom CRDTs.

2.5 Data Replication and Consistency Models

Replication is the process of creating and maintaining multiple copies of a program and its data on machines at different locations. It is employed to increase availability, locality, and data durability. Availability is increased because, if one replica is unavailable, the service can still be accessed from another replica. Locality is better because data can be placed closer to where it is accessed through one of the replicas being closer to its accessors, thereby reducing latency and increasing system throughput. Data is durable because, if one replica experiences a fault, the data can be recovered from another replica. Replication is also used to increase the scalability of the system because more resources can be added to the system by adding more machines.

With replication, the problem of data consistency arises. The network does not guarantee the order in which the messages are delivered across multiple replicas. This can cause the state in replicas to diverge because operations are usually not always commutative, disrupting the functionality of the system. Consistency is a property that ensures that all replicas of the same data are observed and updated in a logical order so that clients have a coherent view of the data. In most cases, it is desirable that all replicas eventually converge to the same state.

There is a limit that must be considered when designing a consistency model for a distributed system. The CAP theorem [24] proves that it is impossible for a distributed system that operates under the asynchronous model and crash fault model to simultaneously provide (strong) consistency, availability, and partition tolerance. Strong consistency (C), which is further explained in Section 2.5.2, ensures that every read operation perceives the most recent write on the system. Availability (A) guarantees that every request to a functioning node receives a response. Partition tolerance (P) means that the system continues to operate correctly even when messages are lost or delayed between nodes. The theorem states that only two of these properties can be simultaneously provided. Because network faults are inevitable and cannot be distinguished from a node crash fault under the asynchronous model, partition tolerance must be guaranteed for a correct system. Therefore, a system can be designed to provide either strong consistency (CP) or availability (AP). The choice between these trade-offs depends on application requirements.

2.5.1 Replication Models

Replication can be performed in different ways, depending on the requirements of the system and data consistency. Replication models define how replicas are organized and

how they communicate with each other and clients to perform data replication. Different models provide different designs with properties that may be more suitable for specific consistency and data persistence requirements. The models are described as follows:

Primary-Backup Client requests are handled by the replicas that apply them, resulting in state mutations. There are replicas in the system that can handle client requests directly. These replicas, the primaries, propagate requests from the client to the other replicas to begin the replication process. There can also be backup (or secondary) replicas that do not handle client requests directly and only receive forwarded requests from the primaries and apply them. This differentiation of primary and backup replicas is useful for modelling the replication process and defining the consistency model. In primary backup [7] replication, there is only one primary, and all other replicas are backups. The least strict model is that in which all replicas can directly apply updates.

Active vs Passive In passive replication [26], the client contacts one replica, and then the request propagation and data replication are performed between replicas. In active replication [58], the client, in addition to issuing its requests, coordinates the replication procedure with replicas. The drawback is that there is a significant computational load on clients that may not scale well with the number of operating replicas, as clients need to be connected and exchange messages with them. Hybrid solutions exist, in which active replication is performed with a subset of replicas.

Synchronous vs Asynchronous Upon receiving an operation, synchronous replication requires that it is synchronized to a subset of the other replicas, before its completion is acknowledged to the client, by waiting for the acknowledgment of the operation from this subset. This ensures that sufficient replicas have registered the operation, but it also increases the latency of operation completion. On the other hand, asynchronous replication does not require that the operation be synchronized with others before it can be acknowledged to the client, and it can be performed as soon as it is applied to one replica. Synchronous replication is used in strong consistency models, and asynchronous replication is used in weak consistency models.

Full vs Partial Data in the system can be fully or partially replicated. With full replication, all data is replicated in all replicas. It is useful when there is no need to localize in which replicas any part of the data resides, all replicas have the full data, and it is always available in every machine. However, this is not scalable because redundant data is stored in all replicas, stored in replicas where certain data points might not be accessed, and operations are always propagated and delivered to all replicas. Partial replication avoids this problem by replicating only subsets of data in each replica using a partitioning scheme, resulting in better scalability and avoiding placing data where it is not (frequently) accessed. This requires the target data to be located in the replica(s) containing it when propagating an operation.

Static vs Dynamic Replicated data must be assigned to replicas in a certain manner. In static replication, this assignment is fixed and configured in advance, and remains unchanged during the operation of the system. This assignment method is simple to

implement, manage, and reason about, but it lacks adaptability to potential changes in the system, such as the addition of new replicas or changes in data access patterns. Dynamic replication solves this problem by allowing the assignment of data to change during the system operation. This allows the system to adapt to changes, enabling load balancing, improving data availability by strategically redistributing data in the case of faults, and optimizing resource usage by placing replicas in nodes that frequently access the data.

2.5.2 Strong Consistency

Strong consistency describes a level of consistency in which all replicas in the system agree on the most recent state of the data at all times. This means that every read operation on any replica returns the state in which the most recent write operation is performed, as if there is a single centralized copy of the data. A system with this consistency ensures that its state evolves linearly in each replica. It is a property required for systems with noncommutative operations to ensure that replicas converge to the same state.

This model comes at the cost of sacrificing availability, as per the CAP theorem [24], meaning that if the network between the replicas is partitioned sufficiently, the system stops responding to ensure that replicas remain consistent. Strong consistency requires synchronous replication, which incurs increased operation latency and limited scalability when compared with weak consistency models.

The linearizability model fits the definition of strong consistency. It provides a real-time guarantee of the behavior of a set of single operations (reads and writes) on a single data object. Under linearizability, a write operation once complete implies that all later reads should return the value generated by that write or a later write. A read operation that returns a certain value also implies that all later reads should return the same value, or the value produced by a later write. This model is tantamount to the state machine replication strategy, where each replica behaves as a deterministic state machine that executes the same operations in the same order.

Consensus solutions can be used to implement strong consistency by implementing state machine replication. For a set of operations, a sequential number is assigned to each instance of consensus and each operation is decided in each instance. If the proposed operation is not decided in the current instance, it is proposed in the next instance. In this manner, only one operation is decided per instance and the operations are ordered by the instance number. Replicas execute operations strictly following this sequence, which is not necessarily the order in which they are decided.

2.5.2.1 Quorum-Based Replication

Quorum-based replication is employed in strong consistency to ensure that operations are only considered complete after being executed in a sufficient subset of replicas. Given a set of replicas $P = \{p_1, p_2, ..., p_n\}$, where n is the number of replicas, a sufficient quorum system (for strong consistency) is given by the set $Q = \{q_1, q_2, ..., q_m\}$ of subsets of P such

that $\forall q_i, q_j \in Q, q_i \cap q_j \neq \emptyset$. This means that any two quorums must intersect to ensure that any two operations are executed in at least one common replica.

Operations can generally be categorized as read and write operations. Reads are commutative with other reads. In this manner, we can define a read-write quorum system as a pair of sets $R = \{r_1, r_2, ..., r_k\}$, $W = \{w_1, w_2, ..., w_l\}$, of subsets of P such that $\forall r_i \in R, w_j \in W, r_i \cap w_j \neq \emptyset$ and $\forall w_i, w_j \in W, w_i \cap w_j \neq \emptyset$. This system has the benefit of allowing smaller read quorums, which allows faster read operations.

Quorums are defined by the number of faulty replicas f that can be tolerated in the system while it continues operation and the desired efficiency of different operations. In particular, $f = min(n - W, n - R)$. Therefore, when planning a system that can tolerate f faulty machines, the number of deployed machines n will depend on the quorum type.

Different types of quorum with different properties can be used. The most commonly used quorum system is the majority quorum with $R = \lfloor \frac{n}{2} \rfloor + 1$ and $W = \lfloor \frac{n}{2} \rfloor + 1$. Every operation must be executed across a majority of replicas ($> \lfloor \frac{n}{2} \rfloor$). It is the best fault tolerant quorum system because it provides the maximum value for f as a function of n, $f = \lfloor \frac{n}{2} \rfloor + 1$, while maintaining the quorum intersection non-empty.

2.5.2.2 Chain Replication

Chain replication is another simple strategy that defines node topology to ensure strong consistency. Replicas are organized in a chain, with the first being the head replica and the last being the tail replica. Each operation is received in the head replica and applied down the chain of replicas until it reaches the tail replica, which is then considered complete. The tail replies to the read operations as it is guaranteed that it holds the most recent stable state of the data. Linearizability is enforced in the system because the operations are executed in the order in which they are received in the head replica and that order is maintained down the chain.

Upon detecting a replica failure, the chain is repaired by either re-linking it if a middle replica fails or by assigning the head or tail role to the next or previous replica. Re-linking involves linking the neighboring replicas of a failed replica. While this process is ongoing, current updates that are being processed are stalled and new updates are queued.

2.5.3 Weak Consistency

Weak consistency relaxes the constraints of strong consistency to provide availability, guaranteeing that the system replies even if some replicas are not reachable. The system also performs better and becomes more scalable, because it does not require the state to have a linear evolution that is perceived globally. Hence, a client can receive its replies as soon as its operations are applied to the replica with which it is in contact, while replication mechanisms are performed in the background (asynchronous replication). To guarantee some level of persistence of operations, this assumption can be slightly relaxed

to require that the operation is applied to a certain number of replicas before replying to the client, sacrificing availability.

Multiple models fit the definition of weak consistency. The model employed depends on the application requirements and trade-off between consistency and performance. The state in the system cannot evolve linearly, owing to its weak consistency. However, it may evolve in a way that makes an underlying semantic sense for the application and its clients.

2.5.3.1 Conflict Resolution

In contrast to strong consistency, weak consistency does not enforce a linear total order for the issued operations. As such, concurrent operations exist, which are operations applied to the system in different replicas in different orders. With a lack of a defined order, concurrent operations can cause conflicts that diverge the state in different replicas, which is an undesirable effect in most applications. Conflicts can occur on concurrent writes that, if applied in different orders, result in different final states, and possibly in reads on replicas that received concurrent writes. Such conflicts must be detected and resolved

To detect conflicts, the system needs to associate metadata to operations and store it alongside the data. Read operations, in addition to retrieving the data value, also retrieve this metadata to the client, and write operations carry the metadata the client previously stored that when arriving at a replica, are used to detect potential conflicts.

For instance, metadata could be a list per data object of the updates that were applied. A conflict between two replicas can be detected by comparing their lists of updates and seeing that neither is an extension of the other, meaning that the replicas received operations concurrently. Although this technique works, it is not scalable because the list of updates grows indefinitely, and it is inefficient to store it, carry it in operations, and compare two large lists of updates.

A more scalable approach is provided using vector clocks [20]. A vector clock V is a list of counters, where each $V[i]$ represents the number of updates applied to a data object at a replica r_i. Each replica maintains its own vector clock V_i and clients store the most up-to-date vector clock that they have observed. This method compresses previous lists of operations by attributing sequence numbers to updates and memorizing only the most recent one. Concurrent updates are detected when perceiving two vector clocks, V and V', such that $\exists i, V[i] > V'[i] \wedge \exists j, V[j] < V'[j]$.

Different techniques can be used to resolve conflicts and reconcile divergences. The simplest one is the last write wins (LWW) policy, where an order is defined among concurrent writes, such as the wall clock time in replicas, the last one in the order is applied to the system, and the others are ignored. This policy is simple, but may not be sufficiently expressive for some applications.

Alternatively, the system can apply concurrent writes, maintain different versions of the state, and return them to the clients when they read the system. The client can then

decide on how to resolve the conflict. This is an application-dependent policy and a more flexible approach, but it requires the client to be aware of the semantics of the data and how to resolve conflicts.

A more robust approach is to use merge procedures that, given the state of two divergent replicas, know how to compute a new state that combines both divergent states deterministically. CRDTs [59] are an implementation of a merge procedure, which are data types designed to be conflict-free by applying commutativity to states or operations.

2.5.3.2 Eventual Consistency

The weakest consistency model is eventual consistency [65]. It does not guarantee any order in which operations are applied to the system, nor does it apply any time constraints on how its state is perceived externally. It only enforces that all replicas eventually converge to the same state in an unbounded time window. This means that if no new updates are performed in enough time, all replicas will have the same state.

Approaches to implement eventual consistency involve propagating updates to all replicas holding the data, and then detecting and resolving conflicts using conflict resolution techniques.

2.5.3.3 Causal Consistency

Causal consistency [37] is a consistency model that ensures that each client always observes a system state that respects the time causality properties regarding the operations they issued. These properties are defined by happens-before relationships, or causal dependencies, between operations, which are relationships that define one operation depending on another. Such dependencies must be executed prior to the operation. However, not all operations are causally related. As such, causal consistency does not enforce a total order of operations but rather a partial order.

Happens-before relationships are typically described by a combination of properties called session guarantees. The strongest form of causal consistency enforces all properties. However, weaker forms can be defined by enforcing only a subset. The session guarantees are as follows:

Read your Writes A client must always be able to observe the effects of all previous updates it issued (and for which it got a reply from the system). Observing these effects implies that the returned value from a client's read operation must reflect the effects of previous write operations issued by that same client.

Monotonic Reads Subsequent read operations issued by the same client should observe either the same state or an inflation of the previous state read, which was modified by new write operations. This property ensures that the state does not regress to the past in the client's perspective.

Monotonic Writes The effects of multiple write operations issued by a given client must be observed respecting the order in which they were issued by every client. This

property ensures happens-before relationships between the write operations issued by the same client.

Writes Follow Reads If a client reads the effects of a write operation in its session, then any subsequent write issued by that client must be ordered after the previously observed write. This is also known as the transitivity property: if write w_1 precedes read r that observes it, then write w_2 after r must also be ordered after w_1, $w_1 \prec r \wedge r \prec w_2 \Rightarrow w_1 \prec w_2$. This property establishes a connection between sessions of different clients, whereas the other properties apply only to a single client's session.

The causality properties do not guarantee that the state of multiple replicas converges to a single value. Divergence can occur and persist forever without breaking any causality. The causal+ consistency model [46] is an extension of causal consistency, which combines the properties of causal and eventual consistency. This ensures that the state of replicas eventually converges to a single value by handling conflicts, while maintaining causality properties.

Causal+ consistency is the strongest of the weak consistency models. To enforce causality properties, clients need to be sticky, meaning that each must only communicate continuously with a single replica. If that replica fails permanently, those clients are forced to restart their sessions with another replica, which might sacrifice some causality properties. This process is called a session or client migration.

It is important to note that causal+ consistency, where dependencies can exist between operations on different data objects, cannot be provided with partial replication when replicas can fail indefinitely [45] without sacrificing some availability to ensure that operations persist in more than a number of replicas, and assuming a maximum number of replicas f will fail.

Implementing causal and causal+ consistency, involves ensuring that for each client, if a value generated by w_1 is observed, no value can be observed that has been overwritten by any w_2 such that $w_2 \prec w_1$. This is done by delaying the effects of w_1 until the effects of w_2 are observed. This requires clients to store a causal history, which is a metadata log of all updates whose effects have been observed in their session. When a client performs an update, its causal history is sent alongside the update and recorded in replicas.

A causal history can be represented in different ways, each with different properties. The most direct and intuitive way of representing dependencies is to model them in a dependency graph, an acyclic directed graph, where each node represents an operation, and each edge represents a dependency between operations. The limitation of this approach is that the graph size and cost of verification of the causal order increase as the number of issued operations increases.

In addition to being used for conflict detection, vector clocks [20] are a popular representation of causal history. They are a compact way of representing the latest version of a data object in a replica. When a replica receives an operation, it verifies whether it can be executed by comparing its vector clock, V_1, with the one sent alongside it, V_2. It can be executed if $\forall i, V_1[i] \geq V_2[i]$, meaning that the replica observed all the updates

that the sender of V_2 observed. Concurrent vector clocks require conflict handling as we discussed before. This approach is possibly limited to systems with a large number of replicas because the vector clock size is proportional to the number of replicas.

A popular method that significantly reduces the size of metadata is to use physical timestamps. In this approach, the system has a global clock in all the replicas. With a single global clock value, operations are coupled with the timestamp value of when they are received in a replica, and can be executed in a replica when it is certain that it will not receive any other operations with a lower timestamp. The problem with this approach is that synchronizing distributed clocks with protocols such as NTP [50] induces some overhead, and even so, there is still a non-null uncertainty interval that can cause causal properties to break. Some systems [17, 1] use additional metadata to untie operations with very close timestamps.

It is also possible to enforce causal consistency by managing how messages are propagated, thereby significantly reducing the use of metadata for storage and messages. These techniques define node topologies that organize replicas in a communication topology where messages are received in an order that ensures causality.

Saturn [6] is a node topology where replicas are organized in a tree. Saturn and similar solutions offer low visibility time, which is the time between an update being issued and being visible to all replicas, without the need for frequent metadata broadcasts. The limitation of this approach is that operations need to traverse the topology and may take time to reach some replicas, which results in higher operation latency caused by waiting for the replica to receive the necessary dependencies and be consistent with the client's past. Topologies may also not be robust to faults. Upon replica failure, the topology may need to be restructured, which is costly and delays the propagation of operations while this process is ongoing.

2.6 Byzantine Fault Tolerance

Distributed systems are vulnerable to process faults, and, as we have seen in the previous section, there are various protocols to deal with these faults. However, the already covered methods only address crash faults, where a process simply stops responding and does not perform any action. Although this is the most common way processes fail, in practice, they can also fail by exhibiting arbitrary behavior. The byzantine fault tolerance model [38] deals with these types of failures and is strictly stronger than the crash fault model.

This model addresses software bugs, memory or disk corruption, machines being overloaded, being unable to respond in time, and malicious behavior of machines controlled by an attacker. Byzantine processes are internal to the system and can be malicious in the sense that they can disrupt the system without any objective of valuable personal gain.

Some common and fair assumptions are considered when dealing with byzantine faults. Akin to the crash fault model, only a subset of machines in the system is faulty. Faulty processes cannot directly change the states of other correct processes. An attacker

can coordinate faulty processes although it is computationally bounded. Cryptographic primitives employed to provide some security properties are assumed impossible to break in a useful amount of time. This assumption is fair given the current state-of-the-art cryptographic schemes.

2.6.1 Byzantine Attacks

The attacks in the byzantine model are performed by processes that are compromised by an attacker. An attacker can be an external entity that has gained unauthorized access to the process already in the system or an internal entity that joined the system intending to disrupt it at some point. These are not external attacks on the perimeter of a process, addressed by other mechanisms such as intrusion detection systems, which are orthogonal to this work.

These attacks involve deviating from the defined protocol behavior to disrupt the correctness or availability of the system. There are many approaches for these attacks, that we enumerate alongside general measures to prevent them.

Masquerading The attacker can impersonate other processes in the system, by using their identity. This can be prevented by using authentication mechanisms, such as digital signatures, that ensure that messages are sent by the process, or the client, who claims to have sent it. Digital signatures are cryptographic primitives that use public–private key pairs to sign messages. The public key is accessible to anyone and is used to verify the signature, whereas the private key is only accessible to the associated process and is used to sign messages.

To ensure the authenticity of the public keys, they are signed by a trusted third party called a certificate authority (CA). The CA is assumed to be secure, or authenticated by other CAs, and the signed public keys it generates are called certificates that are distributed to the processes or sent by the signer process alongside messages.

Message Tampering A faulty process can modify messages it receives. The tampered data may not only be the payload content of the message, but also its metadata. To prevent tampering, messages can be sent alongside an authenticated digest to verify their integrity.

Digest or hash functions are cryptographic primitives that generate a fixed-size and unique byte sequence from target data. If the message or the sent digest is tampered with, its digest will be different from that sent alongside the message. Digital signatures are authenticated digests. Therefore, message integrity can be verified by verifying the signature. If the signature validation fails, the message or signature is tampered with or incorrectly signed. There are also HMAC [4] constructions for authenticating digests.

Message Generation The messages directly generated by a faulty process, which include wrappers over the received messages, can contain different data than that intended by the defined protocol. Some of these messages can be discerned as wrong immediately, depending on the system semantics, and their digital signature can serve as proof of

misbehavior of that process. In other cases, the system must tolerate incorrect messages or use the techniques described in Section 2.6.2.

Storage Tampering Data that is stored in a faulty replica can also be tampered with, by being modified or destroyed. This means it is not sufficient to simply store the data without any integrity verification, because incorrect results can be returned to the client or propagated to other replicas.

Message Omission or Delaying A faulty process can choose to not send messages it is supposed to send, or delay them for an arbitrary amount of time, disrupting the continuous operation of the system. It is an attack that cannot be distinguished from replica crashing or overload. This can be detected by timeouts if a message is expected to be received within a certain amount of time.

Message Replay Messages that are redirected or generated can be repeatedly propagated, with the goal of other replicas processing them twice causing correctness errors. Even if messages are authenticated correctly, they can still be repeated. This can be prevented, for example, by generating a random unique number with the message, a nonce, storing the nonces of messages that were already processed, and ignoring messages that have the same nonce.

Sybil Attack The system expects that the number of faulty processes is bounded and smaller than the number of correct processes. The sybil attack [16] can create multiple identities and join the system with them, thereby increasing the number of faulty processes and invalidating this assumption. The most intuitive approach is to have a centralized entity to validate the entrance of new processes in the system. Blockchains use decentralized mechanisms described in Section 2.6.3.3.

Eclipse Attack An eclipse attack [28] is an attack on the overlay network protocol, where faulty processes isolate a correct process or a subset of correct processes from the rest of the system. This is achieved by having the faulty processes connect to the target correct process(es), filling its neighborhood, thus preventing it from connecting to other correct processes. Consequently, the target process cannot receive or propagate messages from other correct processes. To mitigate this, connections between processes must only be established when validated in such a way that the resulting membership (probabilistically) avoids isolating correct processes.

Collusion Attacks Collusion attacks are performed by multiple coordinated faulty processes to disrupt the operation of the system by exploiting its mechanisms. For example, a quorum can return an incorrect result if sufficient faulty processes collude to do so in that quorum.

Denial of Service Attackers can disrupt the availability of the system by flooding it with requests or exploiting its mechanisms. Denial of Service (DoS) attacks are not byzantine attacks, as they can be performed externally, but they can also be effectively performed internally by faulty processes. As such, when building a byzantine fault tolerant system, it is a valid concern that security mechanisms do not easily exploit vulnerabilities to DoS attacks.

Eavesdropping Byzantine attacks only encompass active attacks that directly disrupt the system by performing actions that attempt to manipulate how the system operates or what information it provides. Passive attacks involve only observing sensitive information inside the system and its messages. These attacks can be performed on messages in transit or stored data. This information can be protected using encryption. However, problems arise when the encryption key is stored, how it is distributed, and if the original key is lost, how is the data recoverable.

2.6.2 Trusted Computing Bases

To attain security properties, a computer system must depend on certain forms of trust. The trusted computing base (TCB) of a system is the smallest amount of code, hardware, people, processes, and/or others that it must trust to meet the desired security requirements. The TCB is always assumed correct to build functionalities on top of it, using the properties it provides. Ideally, a TCB is small and simple, yet robust, to reduce the attack surface and number of entities that need to be trusted.

Trusted computing is a TCB technology that guarantees that a computer will consistently behave in the expected manner for a certain set of computations. These behaviors are enforced by trusted hardware, and thus, the TCB is the hardware root of trust. This involves hardware embedded protection of data and code, such as cryptographic keys and algorithms implemented in the hardware. Trusted computing allows for the optimization of security algorithms, including byzantine fault tolerance algorithms. An example is presented in Section 2.6.3.2.

Trusted hardware is tamper-proof, meaning that the software running in the machine, possibly controlled by a byzantine attacker, cannot tamper with their own trusted hardware, without being detected by others. It mitigates some of the previously mentioned byzantine attacks. This forbids the generation of messages that do not follow the protocol.

Trusted hardware solutions usually incur some overhead when calling their functions, and have limited storage and memory size for computations. In addition to these limitations, there are other reasons to opt not to use trusted hardware, such as its cost and availability in all devices and trust in the hardware manufacturer.

Trusted execution environments (TEE) are a recent trusted hardware solution that is supported in the secure area of a main processor. A TEE is an isolated execution environment that allows the execution of arbitrary code in a secure manner inside the CPU with respect to confidentiality and integrity. In other words, it provides protection against sensitive data exposure and tampering.

2.6.2.1 Intel SGX

Intel SGX (software guard extensions) [12] is a TEE solution that is supported in Intel CPUs and provides a secure isolated execution environment through the use of enclaves. Enclaves are protected environments that contain the code and data of the security sensitive

computation of an application. Their execution is isolated from untrusted software, which includes the rest of the software, operating system, and other enclaves.

The untrusted part of the software calls an enclave method similar to an API, and the results are returned to it. Enclaves are protected from the outside, data read from outside the enclave is returned encrypted, and computations cannot be discerned or inferred. Attacks on this security property have been discovered [55] and as of the date of this work Intel SGX was effectively patched against all of them. Multiple enclaves can exist in the same application. Enclaves are remotely attested, which allows a remote party to authenticate that the software running in the enclave is the expected one. Enclaves should be designed to be small, as they are limited in memory size, and to be the least necessarily called possible, as there is a performance cost incurred when calling enclaves [67].

2.6.3 Byzantine Replication

The problem of data replication with byzantine fault tolerance comes with its own set of challenges. A proper set of protocols needs to ensure that wrong results are not returned to clients (or provide some bounds on how often and for how long this can happen), that correct replicas do not diverge, at least permanently, that consistency guarantees are not broken, and that correct replicas do not become unavailable indefinitely because of the actions of byzantine attackers.

These anomalies can be caused by direct attacks on the application data or by attacks on the metadata, which include attacks on the replication protocol, attacks on the connection structure between replicas (e.g., overlays), and their communication patterns. This means that these protocols must be designed with byzantine fault tolerant mechanisms used to mitigate these attacks under certain system assumptions. These consist of some of the techniques we have described previously, in some cases, using of trusted hardware.

2.6.3.1 Practical Byzantine Fault Tolerance

One of the first implementations of a byzantine fault tolerant replication protocol is practical byzantine fault tolerance (PBFT) [8]. This protocol solves the byzantine consensus problem with no termination guarantees. Therefore, it can be used to implement byzantine replication with state machine replication.

PBFT operation is similar to that of Multi-Paxos [63]. The leader replica receives client requests and proposes an order between them to the other replicas by assigning each request a sequence number, and the replicas execute them sequentially based on this number. Unlike Multi-Paxos, PBFT requires $3f + 1$ replicas to tolerate f faults, f more than Multi-Paxos. This is because, to establish a majority quorum, if f replicas give a false response, another $f + 1$ correct responses are required identify the correct result.

To ensure that the correct replicas execute the same sequence of requests, after receiving the initial proposal, they must confirm that $2f + 1$ of them received the same sequence number for a given request before agreeing to commit to the operation, which is another

step that requires $2f + 1$ confirmations. When the leader fails, detected by $f + 1$ other replicas, the protocol defines a consensus round to change the view, establish a new leader, and continue the protocol.

2.6.3.2 MinBFT

MinBFT [64] is an optimization of PBFT enabled using trusted hardware. Unlike PBFT, it only requires $2f + 1$ replicas to tolerate f faults and one fewer communication phase. This is done by the leader replica using trusted hardware to assign a sequence number, using a secure counter, for each operation proposed.

Using a secure counter implies that a verifiable sequence number will never be assigned to two different messages, and will never assign a number that is not the next in the sequence, even if the leader is compromised. This allows the protocol to have f fewer correct replicas and one fewer communication phase because the other replicas can validate the sequence number by themselves. Replicas can start to confirm that they are committing the operation as soon as they validate the sequence number, which guarantees that the sequence number is correct and that the operation is next in the sequence.

2.6.3.3 Blockchains

Blockchains are decentralized and distributed ledgers designed to keep a record of transactions, groups of operations, while being resilient to byzantine attacks. The blockchain stores signed user transactions that are grouped and inserted into the system in blocks. Each block is validated and signed by a node. The blocks are chained by including the hash of the previous block in the next block. This makes the blocks and their order immutable. Therefore, any mutation in the blockchain negates its validity. In this manner, a single order of transactions is (eventually) established, and the semantic state of the application can be computed deterministically. For each new position in the chain, a block is selected among the nodes using a byzantine consensus protocol.

Bitcoin [54] introduced the concept of blockchain. To decide blocks, it uses a relaxed byzantine consensus algorithm, where transactions can potentially be rolled back, instead of termination not being guaranteed. Blocks are constructed in parallel by multiple nodes, which can result in the concurrent addition of different blocks. This may cause the chain to diverge, branching into multiple chains. This problem is avoided by Bitcoin using proof-of-work, where in order to insert a block in the chain, a node must prove its computational effort by solving a hard crypto-puzzle to generate the block's validity. This avoids sybil attacks because the only way for one entity to control the evolution of the blockchain is to control the majority of the system's computational power. Even so, the chain can diverge, but at a much lower probability and rate. The longest chain, after a number of blocks since the main chain branched, is considered stable and valid. A transaction was confirmed only after it was in a stable block.

2.6.3.4 Byzantine Causal Consistency

Most existing solutions for byzantine fault tolerance typically focus on the strong consistency model among limited numbers of replicas, using a strong protocol such as PBFT, or blockchain approaches that use decentralized consensus algorithms (e.g., proof-of-work) which incur very high write latencies. Byzantine causal consistency aims to provide causal consistency properties, being available and fast, while tolerating byzantine faults.

Faulty replicas generating operations may attack causal consistency by omitting dependencies, depending on unseen operations (adding false dependencies), and creating different messages with the same identifier (sibling generation), which can cause the system to diverge by having dependencies on the identifier satisfied by different operations. These also encompass a combination of the mentioned attacks or the collusion of faulty replicas to perform them.

Existing secure causal consistency solutions rely on byzantine consensus or another trusted decentralized service to reach agreement on a global order of updates [43, 29], chaining hashes of causal dependencies [43, 34, 33], and relying on trusted hardware to generate correct causal dependencies of operations [43]. Some of these techniques are also applied to perform conflict handling in a secure manner [33]. We do not consider solutions that enforce causal order in total order [18]. We aim for solutions that require no total order of operations and thus provide availability. Some of these solutions are described in Section 2.7.

2.6.4 Verifiable Random Number Generation

Verifiable, unpredictable, and unbiased random number generation is an important feature for many services, including byzantine fault tolerance. For example, overlay network protocols in a byzantine setting can use these random numbers to establish connections between nodes with a proof that it was performed in a certified random manner [31, 15]. These random number generators work by gathering entropy, which is the randomness source that should be unobservable and unpredictable. Trusted hardware can be used to generate these random numbers.

In addition to trusted hardware, these generators are implemented in decentralized distributed protocols. These protocols operate in a way that no single node can manipulate the resulting random number, and that it can be independently verified by anyone. Drand [23] is a service that uses this type of protocol to periodically generate random numbers for use by other services.

2.7 Use Cases

In this section, we present use cases that illustrate the concepts presented previously, explaining their main aspects that can be relevant to our work.

2.7.1 DynamoDB

DynamoDB [13] is a data center distributed key-value storage system. It partitions data in its machines by using consistent hashing [32] over a ring structured DHT. Consistent hashing involves hashing the key of a data value and using the hash to directly determine the node that stores it. Each node in the ring is the coordinator for values whose key hashes are in the range of values assigned to that node. The hash space is circular, so the range of a node can wrap, hence, no value in the space is unassigned.

A coordinator is in charge of replicating its values to a number of successors. If the coordinator fails, the next live successor can become a coordinator for these values. Each value has replication factor N which is the number of data replicas that should exist for that value. Clients read and write to the coordinator replica. New nodes can enter the ring by splitting the range of another node.

DynamoDB provides eventual consistency, low latency, and high availability, but may return stale values. The coordinators respond to the client's operations after receiving a number of replies from other replicas that hold the associated value. Vector clocks [37] are used to detect conflicts, and are resolved by clients who detect them in the reads. More precisely, clients read a list containing the values written by the most recent concurrent writes from a replica and resolve the conflict in some manner.

2.7.2 Cassandra

Cassandra [35] is a data center distributed column oriented storage system. In Cassandra, a table is composed of column families. Super column families can contain more column families. Each column or column family is identified by a unique key. Cassandra allows sorting columns based on the value or time of the update. .

Data is partitioned by key in a ring DHT with consistent hashing, similar to DynamoDB. Each coordinator replicates values to other nodes in the ring, but replicas are chosen based on a data center proximity policy regarding nodes within separate racks and data centers. Cassandra provides eventual consistency, resolving conflicts with CRDTs [59], in which column data types are defined as CRDTs.

2.7.3 COPS

COPS [46] is a data center key-value storage system that optimizes geo-replication compared with previous solutions. COPS introduced the concept of causal+ consistency and enforced it using dependency graphs of causal relationships between operations. Conflicts are detected using these graphs and resolved by default with a last-writer-win policy.

Each data center has a COPS cluster similar to that of a DynamoDB ring. Each coordinator replicates its values through a chain of nodes in a cluster. This provides linearizability, which ensures that the resulting system remains linearizable as a whole. Chaining replicas is acceptable because partitions inside a data center are very rare.

Replication between data centers occurs asynchronously and periodically, providing low latency for clients, as they do not need to wait for replicas across data centers. This is also acceptable because it is expected that a client will stick to its closest data center

2.7.4 Legion

Legion [44] extends user-centric Internet services with peer-to-peer connections between clients, enabling them to replicate data from servers and synchronize it among them. The server provides clients with other clients' addresses, and the clients communicate directly through an unstructured overlay network optimized for latency.

Legion offers an extensible library of data types encoded as CRDTs, to enable convergence. Using its gossip protocol for multicasting, the Legion propagates and receives updates in a way that respects the causal order.

2.7.5 Practical Client-side Secure Replication

This solution [43] is implemented as an extension of the Legion service to provide weak consistency semantics in byzantine settings. In this case, it provides a byzantine causal consistency to the replication between clients in Legion.

Operations include a unique number and a hash of all direct causal dependencies. In this way, every operation can only depend on operations that have been previously generated in the system (no future dependencies), and a malicious replica cannot omit the dependencies of an operation in a manner that introduces gaps in the dependency graph.

Upon detection of sibling operations by a replica, it creates a proof of misbehavior against the replicas that generated those operations, and the server is notified to exclude them from the system. In addition, the server periodically sends a summary of its state containing the hash of the last observed operations. Clients can use this summary to verify if they have received the same operations. This summary is also used to determine whether the operations are properly propagated to the system. If not, it means that the client is most likely to be under an eclipse attack and it should contact the server to connect to other clients.

This solution has a version where it ensures operations are generated with its real dependencies. It involves delegating the reception and generation of operations to a trusted service, guaranteeing that causal dependencies are faithfully assigned.

Colluding replicas can still generate incorrect dependencies by creating an operation o_2 that is concurrent with some known operation o_1 when in reality $o_1 \prec o_2$. This problem is addressed by having real-time timestamps uniquely attributed to each operation using a decentralized timestamping service. This establishes an external order of operations. Timestamps can be validated by any replica using the service certificate. This approach requires that operations are defined as undone and redone to be able to enforce the total order of operations when they are received out of order.

2.7.6 Omega

Omega [10] is an edge secure event ordering service that employs Intel SGX. The ordering service runs in edge servers and assigns logical timestamps to events, defining the total order for all events that occur at a given node. These timestamps cannot be tampered with because all events are generated inside the enclave. Omega guarantees integrity, freshness, and causal consistency. It also applies a partial order to events by ordering them per tag.

Periodically, edge servers are attested by the cloud, which checks the integrity of the enclave and freshness of the events. This allows clients to trust the server they are in contact with by checking the cloud certificate on that server.

Events are stored in the untrusted part of the edge server and their integrity is checked by the enclave. A merkle tree [48] containing the hashes of events is used to check integrity, where only the root hash is stored in the enclave. This addresses the enclave's space limitations, being able to check the integrity of the events without having to store all their hashes in the enclave by recomputing the tree and checking if the root with the one in the enclave match. This approach guarantees that edge servers cannot modify application data, return stale data, and modify the causal order of events without being detected.

2.7.7 Engage

Engage [3] is a storage system that offers efficient support for session guarantees in a partially replicated edge setting. It combines the use of vector clocks and distributed metadata propagation services such as Saturn [6] to achieve both low visibility times and low operation latency.

For each object replicated in an edge server, a vector clock is stored with that object. Each edge server also maintains its vector clock, which captures the state version of its local state and encodes the highest version sequence number of operations executed from each edge server. Each client maintains two vector clocks, one for capturing the read on objects and the other for writes. Edge servers hold operations until they are safe to execute, based on the desired session guarantees, the vector clock at the edge server and the vector clock with the operation. Edge servers periodically send to each other their vector clocks to account for different objects they do not own.

Engage's metadata service is organized in a tree, where updates and their associated vector clocks are propagated. The acyclic layout of the service guarantees that every operation that could be a dependency has been delivered locally. Therefore, updates can be immediately applied.

2.7.8 Arboreal

Arboreal [22] is a distributed key-value storage system designed for cloud and edge infrastructures. It adopts a hierarchical tree structure, where the root is the cloud, and the data at a node is the union of the subsets of the data at its children, if any. The cloud

possesses all the data. This also reflects the heterogeneous nature of the edge. One tree is deployed per region, and data is replicated between regions using a geo-replication method in the cloud servers.

The acyclic property of the tree structure guarantees that messages are delivered with respect to happens-before relationships, thus enforcing causal+ consistency. It also prevents the heavy usage of the metadata necessary to enforce causality. If a node does not have the data requested by the client, it requests it from its parent, and this behavior is repeated recursively.

Arboreal allows different data persistence levels. Level number l equals the number of replicas in the branch, and the data must be replicated before returning to the client. The level $l = \infty$ indicates that the data must reach the cloud.

A node with a failing parent connects to its grandparent and synchronizes with it to obtain its up-to-date data. This process does not disable clients from continuing to issue operations at the child server, only waiting if persistence is required. If the node to which the client is connected fails, it migrates to a new node. It can vertically migrate, where the target node waits for updates until it is up-to-date with the client timestamp. It can also migrate horizontally where the target node asks its parent for the client's data. Vertical migration can cause the latency to the client to increase because the node is upper in the tree, and horizontal migration takes longer to establish because the data that the client wants needs to move to the target node.

2.8 Summary

In this chapter, we explained the concepts required to understand our work. We began by explaining the different distribution models, focusing on edge computing, some related distributed abstractions, and how distributed storage systems are composed. We then presented some data models for storage systems and the problem of replication and data consistency, focusing on the properties of causal (and causal+) consistency. Subsequently, we presented the byzantine fault model, its attacks, and some techniques to mitigate them, and how replication can be performed in a byzantine setting. We then presented some use cases that included implemented distributed storage systems, each with different properties and settings. Finally, we conclude the chapter with a discussion of the main challenges that we expect to face while developing this work.

In conclusion, the available literature and work are insufficient to achieve our goal of scalable and efficient replication in edge settings. Existing solutions either do not consider the edge setting, do not provide security against byzantine faults extensively, or do not provide efficient large scale replication. We believe that we can contribute to this research by achieving these goals.

In the next chapter, we detail the specifications of this work, including the main challenges we expect to face, our initial plans we intend to experiment with, how the produced work will be evaluated, and the schedule to achieve these tasks.

3

Future Work

As stated in Chapter 1, the goal of this dissertation is to design and develop an edge-driven storage system resilient to byzantine faults. Our contributions include: i) the design and implementation of a NoSQL data model that includes metadata for security purposes, ii) extending the design and implementing an overlay network for edge setting with security properties, iii) the design and implementation of a storage system with a byzantine replication protocol, and iv) an experimental comparison of our proposal with other state-of-the-art distributed storage systems.

In Section 3.1, we describe the components of the storage system to be designed and implemented, the challenges we expect to face in this process, and the initial plans for the design of each component. In Section 3.2, we describe the metrics we will employ to evaluate the implemented solution comparing with other solutions. Finally, in Section 3.3, we present the estimated schedule for accomplishing the proposed work.

3.1 Proposed Solution

We want to address the problem of efficient and secure data storage at the edge by developing an edge-driven distributed storage system that supports dynamic and partial replication with causal+ consistency in a byzantine setting. Currently, there are no solutions that address this or similar problems. Solutions exist that address provide a replication protocol with these properties in the edge, Engage [3] and Arboreal [22], Omega [10] provides causal consistency with byzantine fault tolerance but does not address replication, and Legion's [44] secure extension [43] only addresses replication on clients.

Achieving these goals presents some challenges, as described below.

3.1.1 Secure Membership in the Edge

The first challenge is the design of an overlay network that is robust to malicious nodes trying to take advantage of the overlay to disrupt the communication between nodes, namely, eclipse attacks, and thus disrupt the system's operation.

Solutions exist in which these overlay networks probabilistically guarantee that there is a connected path between all correct nodes. If a sufficient number of correct nodes exist in the system, this is sufficient to reliably propagate messages across the correct nodes in the system, avoiding eclipse attacks, and making it possible to causally relate operations [51]. Some of these include DHTs such as Fireflies [30, 31], which rely on random numbers, and another [2] which relies on random walks.

The goal is to design an overlay network on top of an existing network that has the aforementioned security properties, and adapt it to the edge environment and the replication model to some extent, perhaps by combining with the properties of edge-driven overlays [11]. Ideally, it should be robust to churn, and the gossip protocol should be inexpensive to the network, reducing message redundancy (receiving repeated and irrelevant messages). This implies that nodes are grouped and close to each other, being able to discern the messages they need. This property is particularly important for achieving partial replication. A partially structured overlay fits these requirements better.

3.1.2 Enforcing Byzantine Fault Tolerance

The second and most important challenge is the development of byzantine fault tolerance mechanisms. Enforcing causal+ consistency in a byzantine setting involves preserving the causal history by detecting if it was tampered with. Solutions that implement causality by propagating operations through a tree [6, 3, 22] are susceptible to eclipse attacks because nodes have a low number of neighbors (one parent). Legion's extension either hashes causal dependencies (which does not defend against fully omitting them), uses Intel SGX [12] to create correct dependencies, or uses a trusted decentralized service to order operations. Ideally, our solution will rely on metadata to enforce causality, perhaps using an approach similar to Legion's extension and SGX solutions.

Allocating data to enable dynamic and partial replication will involve obtaining data that clients need while notifying the system of the new data location.

Stored data integrity must be verifiable in some way so that the entity retrieving it can check its validity. Omega utilizes an Intel SGX to store a verifiable log of operations. Another possible solution is to authenticate stored data.

Correct replies to clients could perhaps be implemented by having a combination of quorum replica answers and checking the integrity of the replied data.

3.1.3 Selecting the Data Model

Another challenge is to select a data model that is suitable for the replication protocol. It should be sufficiently rich to provide useful semantics for various applications. NoSQL data models are better suited to weak consistency and scalability [27]. We plan to use either a document based or a column oriented data model. The former is better at providing more complex entities whereas the latter is better at providing aggregation.

Ideally, the replication model should be agnostic to the data model or even to the API, so swapping different data models is possible and does not require alteration of the former. The only difference would in certain operations is how the data is fetched, locally, or remotely.

3.1.4 Data Confidentiality

A relevant challenge is how to provide data confidentiality in the face of eavesdropping on communication channels, edge servers, or even the cloud. In some cases, certain application data stored in the system is sensitive and must be encrypted. The granularity with which the data is encrypted will depend on the data model.

The main problem is where the encryption keys are stored. The most obvious solution is to make clients responsible for their own key, which is a fair strategy owing to password-based encryption. Other solutions [5] involve partitioning the key into different machines and then acquiring them when required.

3.2 Evaluation

To demonstrate that our solution works and performs as desired, we plan to test it and compare it with other alternatives through experimental comparisons. We plan to use a network emulation to test our solution.

The experimental work will be separated into evaluating the new overlay network, separating it from the system, and evaluating the system as a whole. Sufficient and appropriate metrics will be used to evaluate the performance of both.

3.2.1 Overlay Evaluation

For the overlay evaluation, the goal is to calculate the overhead of the security mechanisms, compared to other overlays with similar properties, although they lack security properties. We plan to measure the propagation reliability (what portion of target replicas messages reach), redundancy (how many messages are repeatedly or undesirably delivered), and latency of gossip messages.

3.2.2 System Evaluation

To evaluate the distributed storage system, we plan to compare it with other systems that offer no security and operate preferably in the same setting (e.g., Arboreal). As such, we plan to measure the operation latency, operation throughput, and data visibility, and see how these evolve when the system is under attack by a variable number of nodes. In addition, we plan to measure, storage, and metadata overheads.

3.3 Schedule

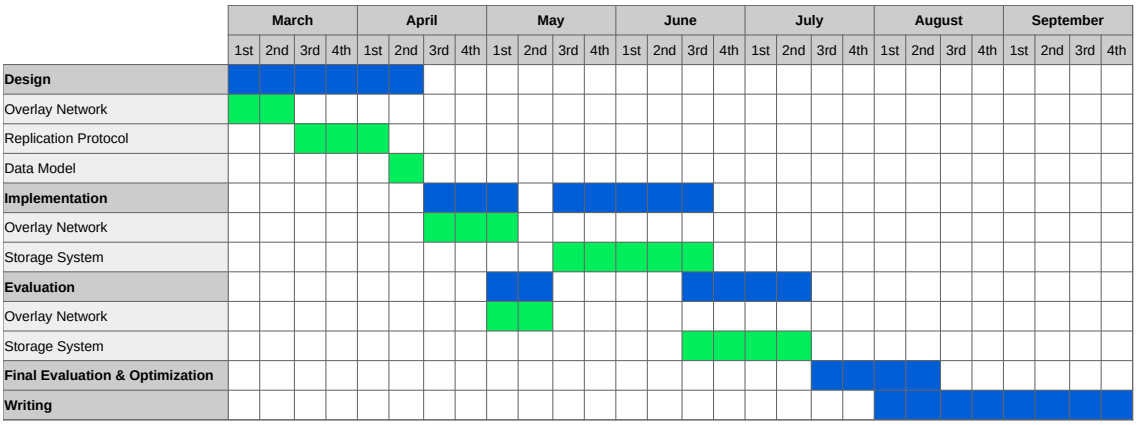

Figure 3.1: Gantt chart with the estimated work schedule.

Figure 3.1, provides an estimated schedule for accomplishing the following main tasks, in the critical path of this work.

Design of the storage system components, including the overlay network, the secure replication protocol, and the data model, following Section 3.1. Some bibliographic refinements are also expected.

Implementation of the overlay network and then the storage system as a whole.

Evaluation of the the overlay network and the storage system, comparing with other works, according to the metrics described in Section 3.2. This task is concurrent with the implementation of their respective parts.

Final Evaluation & Optimization: Minor optimizations of the final solution and their respective evaluations.

Writing of the dissertation.

Bibliography

[1] D. D. Akkoorath et al. "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 (cit. on p. 21).

[2] J. Augustine, S. Chatterjee, and G. Pandurangan. "A fully-distributed scalable peer-to-peer protocol for byzantine-resilient distributed hash tables". In: *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 2022, pp. 87–98 (cit. on p. 33).

[3] M. Belém et al. "ENGAGE: Session Guarantees for the Edge". In: *2022 International Conference on Computer Communications and Networks (ICCCN)*. IEEE. 2022, pp. 1–10 (cit. on pp. 11, 30, 32, 33).

[4] M. Bellare, R. Canetti, and H. Krawczyk. "Keying hash functions for message authentication". In: *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*. Springer. 1996, pp. 1–15 (cit. on p. 22).

[5] A. Bessani et al. "DepSky: dependable and secure storage in a cloud-of-clouds". In: *Acm transactions on storage (tos)* 9.4 (2013), pp. 1–33 (cit. on p. 34).

[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*. 2017, pp. 111–126 (cit. on pp. 21, 30, 33).

[7] N. Budhiraja et al. "The primary-backup approach". In: *Distributed systems* 2 (1993), pp. 199–216 (cit. on p. 15).

[8] M. Castro, B. Liskov, et al. "Practical byzantine fault tolerance". In: *OsDI*. Vol. 99. 1999. 1999, pp. 173–186 (cit. on p. 25).

[9] H. Cloud. "The nist definition of cloud computing". In: *National Institute of Science and Technology, Special Publication* 800.2011 (2011), p. 145 (cit. on p. 4).

[10] C. Correia, M. Correia, and L. Rodrigues. "Omega: a secure event ordering service for the edge". In: *IEEE Transactions on Dependable and Secure Computing* 19.5 (2021), pp. 2952–2964 (cit. on pp. 30, 32).

[11] P. A. 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 (cit. on pp. 8, 33).

[12] V. Costan and S. Devadas. "Intel SGX explained". In: *Cryptology ePrint Archive* (2016) (cit. on pp. 24, 33).

[13] G. DeCandia et al. "Dynamo: Amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220 (cit. on pp. 1, 28).

[14] A. Demers et al. "Epidemic algorithms for replicated database maintenance". In: *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. 1987, pp. 1–12 (cit. on p. 8).

[15] D. Dolev, E. N. Hoch, and R. Van Renesse. "Self-stabilizing and byzantine-tolerant overlay network". In: *Principles of Distributed Systems: 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings 11*. Springer. 2007, pp. 343–357 (cit. on p. 27).

[16] J. R. Douceur. "The sybil attack". In: *International workshop on peer-to-peer systems*. Springer. 2002, pp. 251–260 (cit. on p. 23).

[17] J. Du et al. "Gentlerain: Cheap and scalable causal consistency with physical clocks". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2014, pp. 1–13 (cit. on p. 21).

[18] S. Duan, M. K. Reiter, and H. Zhang. "Secure causal atomic broadcast, revisited". In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2017, pp. 61–72 (cit. on p. 27).

[19] P. DuBois. *MySQL*. New riders publishing, 1999 (cit. on p. 1).

[20] C. J. Fidge. "Timestamps in message-passing systems that preserve the partial ordering". In: (1987) (cit. on pp. 18, 20).

[21] M. J. Fischer, N. A. Lynch, and M. S. Paterson. "Impossibility of distributed consensus with one faulty process". In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382 (cit. on p. 9).

[22] P. Fouto, N. Preguiça, and J. Leit ao. "Large Scale Causal Data Replication for Stateful Edge Applications". 2023 (cit. on pp. 11, 30, 32, 33).

[23] N. Gailly, P. Jovanovic, and B. Ford. *Drand: Distributed randomness beacon*. Tech. rep. DEDIS Lab and DFINITY, 2017. URL: https://drand.love/ (cit. on p. 27).

[24] 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 (cit. on pp. 14, 16).

[25] R. Greenwald, R. Stackowiak, and J. Stern. *Oracle essentials: Oracle database 12c*. " O'Reilly Media, Inc.", 2013 (cit. on p. 1).

[26] R. Guerraoui and A. Schiper. "Fault-tolerance: from replication techniques to group communication". In: *IEEE computer* 30.6 (1997), pp. 68–74 (cit. on p. 15).

[27] J. Han et al. "Survey on NoSQL database". In: *2011 6th international conference on pervasive computing and applications*. IEEE. 2011, pp. 363–366 (cit. on p. 33).

[28] E. Heilman et al. "Eclipse attacks on {Bitcoin's}{peer-to-peer} network". In: *24th USENIX security symposium (USENIX security 15)*. 2015, pp. 129–144 (cit. on p. 23).

[29] K. Huang et al. "Byz-GentleRain: An Efficient Byzantine-Tolerant Causal Consistency Protocol". In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by C. Johnen, E. M. Schiller, and S. Schmid. Cham: Springer International Publishing, 2021, pp. 495–499. ISBN: 978-3-030-91081-5 (cit. on p. 27).

[30] H. Johansen, A. Allavena, and R. Van Renesse. "Fireflies: scalable support for intrusion-tolerant network overlays". In: *ACM SIGOPS Operating Systems Review* 40.4 (2006), pp. 3–13 (cit. on p. 33).

[31] H. D. Johansen et al. "Fireflies: A secure and scalable membership and gossip service". In: *ACM Transactions on Computer Systems (TOCS)* 33.2 (2015), pp. 1–32 (cit. on pp. 27, 33).

[32] D. Karger et al. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 1997, pp. 654–663 (cit. on p. 28).

[33] M. Kleppmann. "Making crdts byzantine fault tolerant". In: *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*. 2022, pp. 8–15 (cit. on p. 27).

[34] M. Kleppmann and H. Howard. "Byzantine eventual consistency and the fundamental limits of peer-to-peer databases". In: *arXiv preprint arXiv:2012.00472* (2020) (cit. on p. 27).

[35] A. Lakshman and P. Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS operating systems review* 44.2 (2010), pp. 35–40 (cit. on pp. 1, 28).

[36] L. Lamport. "The part-time parliament". In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–170 (cit. on p. 9).

[37] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications* (1978) (cit. on pp. 9, 19, 28).

[38] L. LAMPORT, R. SHOSTAK, and M. PEASE. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401 (cit. on p. 21).

[39] A. Langley et al. "The quic transport protocol: Design and internet-scale deployment". In: *Proceedings of the conference of the ACM special interest group on data communication*. 2017, pp. 183–196 (cit. on p. 7).

[40] J. Leitão, J. Pereira, and L. Rodrigues. "Epidemic broadcast trees". In: *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE. 2007, pp. 301–310 (cit. on p. 8).

[41] J. Leitão, J. Pereira, and L. Rodrigues. "HyParView: A membership protocol for reliable gossip-based broadcast". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE. 2007, pp. 419–429 (cit. on p. 8).

[42] J. Leitão et al. "Towards enabling novel edge-enabled applications". In: *arXiv preprint arXiv:1805.06989* (2018) (cit. on pp. 1, 6).

[43] A. van der Linde, J. Leitão, and N. Preguiça. "Practical client-side replication: Weak consistency semantics for insecure settings". In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 2590–2605 (cit. on pp. 27, 29, 32).

[44] A. van der Linde et al. "Legion: Enriching internet services with peer-to-peer interactions". In: *Proceedings of the 26th International Conference on World Wide Web*. 2017, pp. 283–292 (cit. on pp. 29, 32).

[45] A. van der Linde et al. "On combining fault tolerance and partial replication with causal consistency". In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. 2020, pp. 1–5 (cit. on p. 20).

[46] W. Lloyd et al. "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 (cit. on pp. 20, 28).

[47] P. Maymounkov and D. Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65 (cit. on p. 8).

[48] R. C. Merkle. "A digital signature based on a conventional encryption function". In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378 (cit. on p. 30).

[49] D. Merriman. *MongoDB*. MongoDB, Inc., 2007. URL: https://www.mongodb.com (visited on 2024-01-24) (cit. on p. 1).

[50] D. L. Mills. *Computer network time synchronization: the network time protocol*. CRC press, 2006 (cit. on p. 21).

[51] A. Misra and A. D. Kshemkalyani. "Detecting Causality in the Presence of Byzantine Processes: There is No Holy Grail". In: *2022 IEEE 21st International Symposium on Network Computing and Applications (NCA)*. Vol. 21. IEEE. 2022, pp. 73–80 (cit. on p. 33).

[52] C. Modi et al. "A survey of intrusion detection techniques in cloud". In: *Journal of network and computer applications* 36.1 (2013), pp. 42–57 (cit. on p. 1).

[53] G. Mühl. "Large-scale content-based publish-subscribe systems". PhD thesis. Technische Universität, 2002 (cit. on p. 9).

[54] S. Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: *Decentralized business review* (2008) (cit. on p. 26).

[55] A. Nilsson, P. N. Bideh, and J. Brorsson. "A survey of published attacks on Intel SGX". In: *arXiv preprint arXiv:2006.13598* (2020) (cit. on p. 25).

[56] J. Postel et al. *Transmission Control Protocol*. RFC 793. RFC Editor, 1981-09. URL: https://www.rfc-editor.org/rfc/rfc793.txt (cit. on p. 7).

[57] S. Sanfilippo. *Redis*. Redis Labs, 2009. URL: https://redis.io (visited on 2024-01-24) (cit. on p. 1).

[58] F. B. Schneider. "Replication management using the state-machine approach". In: *Distributed systems* 2 (1993), pp. 169–198 (cit. on p. 15).

[59] M. Shapiro et al. "Conflict-free replicated data types". In: *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer. 2011, pp. 386–400 (cit. on pp. 13, 19, 28).

[60] W. Shi and S. Dustdar. "The promise of edge computing". In: *Computer* 49.5 (2016), pp. 78–81 (cit. on pp. 1, 5).

[61] R. Taft et al. "Cockroachdb: The resilient geo-distributed sql database". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 1493–1509 (cit. on p. 1).

[62] *TaRDIS 'Trustworthy And Resilient Decentralised Intelligence For Edge Systems' project website*. 2023. URL: https://www.project-tardis.eu/ (visited on 2023-12-18) (cit. on p. 3).

[63] R. Van Renesse and D. Altinbuken. "Paxos made moderately complex". In: *ACM Computing Surveys (CSUR)* 47.3 (2015), pp. 1–36 (cit. on pp. 9, 25).

[64] G. S. Veronese et al. "Efficient byzantine fault-tolerance". In: *IEEE Transactions on Computers* 62.1 (2011), pp. 16–30 (cit. on p. 26).

[65] W. Vogels. "Eventually consistent". In: *Communications of the ACM* 52.1 (2009), pp. 40–44 (cit. on p. 19).

[66] Y. Xiao et al. "Edge computing security: State of the art and challenges". In: *Proceedings of the IEEE* 107.8 (2019), pp. 1608–1631 (cit. on p. 1).

[67] C. Zhao et al. "On the performance of intel sgx". In: *2016 13Th web information systems and applications conference (WISA)*. IEEE. 2016, pp. 184–187 (cit. on p. 25).