Robust and Strongly Consistent Distributed Storage Systems

Andria Trigeorgi
University of Cyprus, Nicosia, Cyprus

October 2022

Abstract

EMC Digital Universe [1] estimates that the data in the digital universe doubles in size every two years. According to IDC (International Data Corporation) [2], the amount of data in the world was 26 Zettabytes (ZB) in 2017, and 64.2 ZB in 2020. The growth in 2020 was higher than expected because of the COVID-19 pandemic, as more people worked, learned, and entertained themselves online from home. By 2025, IDC says the amount of data will grow to 175 ZB. As more and more data are generated at a high speed, the challenges posed by handling Big Data gain an increasing importance. To cope with this explosion of data, large-scale infrastructures are needed. A Distributed Storage System (DSS) is an infrastructure that can split data across multiple servers.

The design and implementation of DSSs involves many challenges due to the fact that the users and storage nodes are physically dispersed. Some major problems are how to ensure consistency of the sharing data, how to provide concurrent access to the same file for several users at the same time and how to mask hosts failures by allowing new hosts to join, and failed hosts to be removed without service interruptions.

In this comprehensive report, we focus on scalable and efficient data management for DSSs. We propose a set of principles for designing large-scale DSSs. We give a brief overview of consistency models. Then, we describe existing Distributed Shared Memory emulations and Distributed Storage Systems. In particular, we study focusing on the main features which the most DSSs required for Big Data.

Contents

1 Introduction

Applications, social media websites and browsers allow users to create and exchange an incredible amount of user-generated data. The key challenge for many organizations and companies is to design an efficient storage system to cope with this data explosion. A Distributed Storage System (DSS) [3, 4] is a client-server application that allows clients to access and process such a massive amount of data remotely as if they were stored locally.

There are various attributes that have to be taken into consideration while designing a DSS. In addition to the functionality of a classical local system (e.g., storing, retrieval, security etc), a DSS supports data survivability and system availability. Data replication, in which the data are copied on multiple storage locations, is a well known technique to cope with these issues. A main challenge due to replication, caused when the shared data are accessed concurrently, is data inconsistency.

Numerous platforms prefer high availability over consistency, due to the belief that strong consistency will burden the performance of their systems. As a result, they devise strategies to address the issue of consistency, but they rely on system coordinators to provide weaker consistency guarantees. However, modern storage systems attempt to find the balance between the consistency of the data and the availability of the system. How much consistency is good-enough depends on the needs of an application. Applications with weaker consistencies are usually easier to implement and offer lower latency at the risk of returning stale data. While, on the other hand, strong consistency is generally hard to implement but provides up-to-date data at the cost of higher latency.

Fault tolerance [3] is another important property of distributed systems. It is the ability of a system to continue to be both available and reliable in the presence of failures. The two types of failures that are most studied in distributed systems are crash failures and Byzantine failures. A crash failure occurs when a node halts once, and then stops responding completely, and becomes unresponsive (aka, it crashes). In synchronous systems, it is possible to detect a crash failure (using heartbeat or timeout). Fischer, Lynch and Paterson [5] presented the FLP Impossibility Theorem which states that it is *impossible* in an asynchronous system for a set of nodes to reach agreement if at least one node fails even by crashing; asynchrony prevents distinguishing between a process that has crashed and a process that is running slow. Byzantine failures [6] are more complicated because the faulty unit can continue to be active and return incorrect/corrupted values, or not respond at all. In particular, a Byzantine node may arbitrarily deviate from its specified protocol. This can be because of faults in the system (hardware, software bugs, etc.) or because of malicious intent. Byzantine faults are obviously the most difficult kind of failure to deal with because the behavior is non-deterministic and the node might appear healthy to the rest of the network.

The problem of keeping copies consistent becomes even more challenging when failed servers need to be replaced or new servers are added. Since the data of a DSS should be accessible immediately, it is imperative that the service interruption during a failure or a repair should be as short as possible. To this respect, the design of a server reconfiguration service for dynamic networks has became an active area of research [7].

Chapters Overview: In Chapter 2 we propose and discuss a set of principles for designing large-scale Distributed Storage Systems (DSS). In Chapter 3 we present the different consistency models and provide a brief comparison of them. In Chapter 4 we review existing Distributed Shared Memory emulations, discussing their strengths and weaknesses. We focus on crash-tolerant atomic memory algorithms (in either static or dynamic setting) and Byzantine-tolerant atomic memory algorithms. In Chapter 5 we discuss the architecture and the main characteristics of prominent Distributed Storage Systems while presenting the strengths and weaknesses

of each of them. We conclude in Chapter 6.

2 Design Goals of Distributed Storage Systems

We start by discussing three main storage formats [8]: file storage, block storage, and object storage. Each format organizes and presents data in a different way, with its strengths and limitations. Block storage stores fixed-size chunks of data into arbitrarily arrangement (ideal for enterprise databases). Object storage stores data as objects in scalable containers (ideal for unstructured big data). Finally, file storage is the oldest approach to storage; it stores data as a hierarchy of files in folders. A traditional file system [9] is the component of an operating system which is responsible for organizing and retrieving files. The data is maintained on a single storage which is a single point of failure. If the server fails, the data can not be retrieved. A file system defines how files are named, where to store the data, how to retrieve them from the storage, and ensure privacy, preventing accidental or malicious damage. Common file systems are NTFS, exFAT, FAT32, and ext4.

The traditional file systems cannot handle the new requirements in terms of volume of data, high performance, fault-tolerance, and improved capabilities. So distributed storage systems (DSS) took place to cover the need of a shared storage between separate systems, provide a scalable storage to serve thousands of servers, and improve the fault-tolerance that allows the system to continue operating properly when some of its servers fail [3]. Thus, a distributed storage system has much better availability and data durability than a centralized system could achieve.

A DSS [3, 10, 11] is like a "normal" storage system, but the main difference is its environment. As previously stated, a traditional file system has all the data and all the users on a single machine. In contrast, a DSS is a system that allows multiple users to share data through a network in a secure and reliable way. So data is replicated in different nodes and clients are still able to read the data if any node fails. Another difference between the two is the reading time. The time taken by a traditional file system to read the data from the storage is less as it needs a read call to local storage and local processing time. However, the time taken by DSS to read the data from the shared storage is more as there are network remote calls, local data reads to disc and managing the data from multiple servers.

Most systems are built following a client-server architecture, where data is distributed across multiple storage nodes (Storage Servers). However the clients remain unaware of the actual location on which their data is stored. They see an interface that is similar to the interface of a local system. The architecture of a distributed system varies depending on the technological choices based on what the system promises to do. Nevertheless, the architecture must follow some basic rules, which are required for the survival of such large-scale systems. Below we present the requirements that a distributed storage system must support and discuss the challenges that one might face during the design and implementation [3, 10]:

Transparency is an important aspect of a distributed system to hide from the user its distributed nature, appearing and functioning as a normal centralized system. The most important types of transparencies are Access, Location, Relocation, Replication, Concurrency and Failure transparency. When **access** transparency is provided, the clients must be unaware that files are distributed and can access them in the same way as local files are accessed. **Location** transparency refers to the fact that the clients have no indication about the physical location of a resource in the system. The system can be able to hide from the client that an object may be moved to another location. If that movement occurs while the object is being accessed, then the **relocation** transparency is provided. Otherwise, **migration** transparency is provided. **Replication** transparency deals with the hiding from the clients that a resource is replicated

on multiple servers. When a system provides **concurrency** transparency, it is able to hide from clients that other clients are using a shared resource. Finally, its is important that a distributed system provides **failure** transparency. This means that if a server fails, the clients should never notice the failure.

Scalability is an important design goal for distributed systems. It is the ability of the system to handle tasks as the system grows in size. Scalability can be measure along three dimensions: **size** scalability, **geographically** scalability, and **administrative** scalability. The first scalability dimension refers to the users and resources. The growth of both the attributes should be easy and not seriously disrupt the service. A geographically scalable system is the one that can continue function efficiently no matter the distance between the users and resources. Finally, an administrative scalable system is the one that can be easy to manage even if it spans independent administrative organizations.

So, all these scalability types must be met without costing system performance or interrupting the system. However, we have to deal with the limitations of each system. There are two main limitations: centralization and synchronous communication. Imagine that we have a centralized system, that is, a single server. If it suddenly received a much higher number of requests than it could really serve, it would run into its computational limitation. Also if the system has to execute a large number of operations to a single database at the same time, it may exceed the bounds of its storage limitations; only a specific amount of data could be handled at the same time. Finally, there is the issue of the network. If something goes wrong between the communication of the server and a user, the entire system could become delayed or even worse unavailable, from the user's perspective.

Fault tolerance is an important issue in designing a distributed storage system. It is the ability of the system to continue operating in the event of a partial failure without seriously affecting the overall performance. Fault tolerance provides four main features in distributed storage systems, availability, reliability, safety, and maintainability. **System availability** is the probability that a system is not failed and remains operational at any time in order to be used by its users. Thus, a highly available system is one that will most likely be working at any time. **Reliability** is the probability that a system is available to response. A highly-reliable system is one that will most likely continue to perform its core functions without service interruptions, errors, or significant performance decrease during a relatively long period of time. **Safety** refers to the absent of catastrophic consequences on the users and the environment. Finally, **maintainability** is the ability of the system to be repaired and accept modifications.

Data survivability is the capability of the system to prevent loss of data despite multiple disruptions. Without data, most businesses can not operate. For example, a business dealing with sales, bank details, payrolls depend on a network of quickly accessible data. Data survivability and system availability are often achieved by redundant hardware, data replication and IT (Information Technology) maintenance methods that assure network integrity after modifications to the architecture of data center.

Data replication is the process of storing the same data across multiple nodes. With data replication, we avoid having a single centralized metadata server, which then provides a single point of failure. There are multiple benefits of data replication. First, data are replicated to improve data survivability. Thus if the system has a faulty replica, the data can be accessed from a different replica. Also, by maintaining multiple copies, it is possible to improve the performance (including availability) of the system. Thus data replication improves the system performance by dividing the workload among the processes accessing the data; this is important in the case of size scaling. Also, replication is important in the case of geographically scalability, as placing the same data in multiple locations can decrease the access time, since required data may be closer to the process using them.

On the other hand, there are some challenges to maintain consistent data across different locations. A main challenge due to replication, caused when the shared data are accessed concurrently, is data inconsistency. Numerous platforms prefer high availability over consistency [3], due to the belief that strong consistency will burden the performance of their systems. However, modern storage systems attempt to find the balance between the consistency of the data and the availability of the system [12]. Also keeping copies of the same data in multiple locations leads to higher storage and processor costs.

Performance is measured as the average amount of time needed to satisfy client requests. This time includes CPU time, time for accessing secondary storage and communication overhead (i.e., network access time). It is desirable that the performance of a distributed storage system be comparable to that of a centralized storage system.

Storage efficiency is the ability to store the maximum amount of data while consuming the least amount of space and causing the least impact on performance. Thus, if the data is replicated, you also want efficiency in your replication technique to avoid waste storage efficiency and also improve the network latency. There are multiple technologies which try to improve the storage efficiency. One of them is data deduplication [13], which can help to reduce the redundant data on storage. When data is written to the system, it scans all the incoming blocks and creates fingerprints (hash values) for each of the blocks. Then it identifies unique blocks, comparing the previous calculated fingerprints to all the fingerprints within the volume. If the comparison detects identical data, the pointer to the data block is updated, and the duplicate block is removed. Another technique that is present in many storage systems is snapshots [14]. A snapshot is a copy of the data at a specific time. Modifications cause the creation of new versions of snapshots. The new snapshot contains the modified data. The previous version remains in the system, whereas the new version is composed of the previous version and the change.

Multiple users request **concurrent access** to shared data. Thus the system must use some form of concurrency control mechanism to manage and synchronize these concurrent accesses. Many storage systems use locking to guard concurrent access to the same storage. Only the user that has the lock can get access to the storage, while subsequent users get an "access denied" error. However locking an entire storage to change a part of it can halt the access to this storage for an indeterminate duration. Therefore, modern systems use block level locking, which allows concurrent access to the storage by multiple clients. Also there is the byte range locking, which allows locking byte ranges instead of whole blocks. **Data striping** is the process of dividing each object into blocks and distributing the blocks across multiple storage nodes. This technique is used to improve the performance of reads and writes. Another solution for controlling concurrency is based on a versioning mechanism. **Versioning** is a mechanism that allows multiple processes to access and modify shared objects in parallel. The system records versions of changed objects. Each update cause the creation of a new and unchangeable version of the object. In this way, versioning improves concurrent data access, whether versions are changed by the storage service or by the application.

These mechanisms provide different data consistency while allowing concurrent access to the files by multiple application nodes. **Consistency Semantics** determine how the modifications of data is noticeable by other users. A Consistency Model is a contract between processes and the data store. That means that if processes agree to obey certain rules, the store promises to work correctly. For example, when a process performs a read operation on a data item, it is expected to return the value resulted from the latest write operation. These rules should not be very restrictive allowing fast executions and concurrent operations, if this is possible. The study of such consistency semantics is the subject of the next chapter.

3 Consistency Models

A Distributed Storage System (DSS) [3] provides data survivability and system availability. As discussed in the previous chapter, data replication on multiple storage locations is a well-known technique to enhance reliability and improve system performance. Replication can improve reliability by introducing redundancy. In the event of a failure, this means preventing data loss in the network. Also, data replication improves performance by increasing the number of servers available to handle requests and thus dividing the work. Replicating data across different geographical locations can also improve performance by placing a copy of data closer to the process using them. As a consequence, the time to access the data decreases.

A major downside to replicating data is that it introduces the problem of keeping replicas consistent. This means that all copies of data should be up-to-date, otherwise the replicas will no longer be the same. In this chapter, we present the different consistency models from strongest to weakest based on their applicability. Each model has its own rules which restrict the values that a read operation can return. The models with less strict rules are easier to use, but the ones with stricter rules have better efficiency. Thus, a system has to find the best trade-off point between the consistency of the data and the availability of the system based on its needs.

This chapter introduces a novel viewpoint on different consistency models utilized in distributed systems. As mentioned above, a consistency model is responsible for managing the state of shared data for distributed shared memory systems. A shared memory system is a system that consists of processes that concurrently access a shared memory consisting of registers. A process can atomically access a register in the shared memory through a set of predefined operations. The most common operations when discussing about consistency models and how these are affected are the Write or $W_x(a)$ and the Read or $b = R_y$ operations. These define that value a will be written to object (register) x and that value b is returned to process y respectively.

3.1 Strict Consistency

The strict consistency model [3], also known as Unix consistency, is the strongest consistency model. A shared-memory system is said to support the strict consistency model if any read on data item (register) "x" always returns the value written by the most recent write on "x", regardless of where the processes performing the operations. With this type of consistency, all writes are instantaneously visible to all processes. The problem with strict consistency is that it relies on absolute global time. This is impossible to implement in a large-scale distributed system.

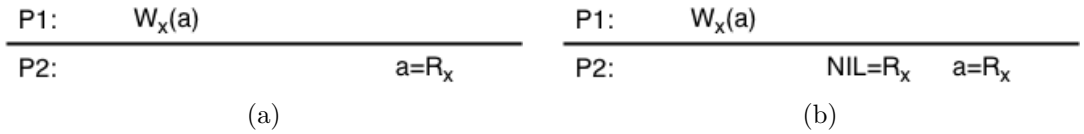


Figure 1: Strict Consistency Model

Fig. 1a depicts a strictly consistent store in which process P1 performs a write operation $W_x(a)$ and then P2 reads the value a. However, Fig. 1b shows a store that is not strictly consistent, since P2 reads the value NIL and some time after that it reads the value a.

3.2 Linearizability/Atomic Consistency

The Linearizability model [15, 3] is weaker than the strict consistency. A shared-memory system supports linearizability if all processes see all shared accesses in the same order. Accesses are ordered according to a global logical timestamp (e.g., an integer value). If the timestamp of an operation $op1$ is less than the timestamp of an operation $op2$, i.e., $t_{op1} < t_{op2}$, then the $op1$ must occur before $op2$ in the sequence seen by all processes. This model provides the illusion that operations happen in a sequential order.

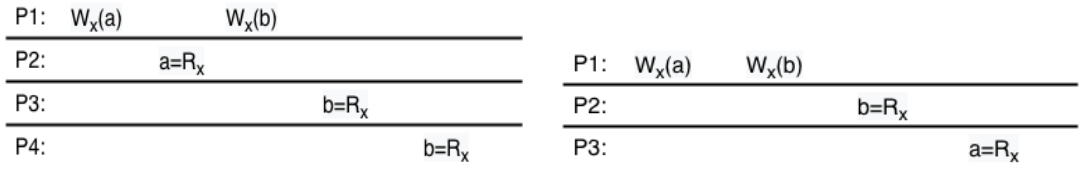


Figure 2: Atomic Consistency Model

Fig. 2a depicts a valid sequence of events for Linearizability consistency. Process P2 reads the value written by its preceding write. P1 completes $W_x(a)$ before $W_x(b)$, therefore process P3 has the value b. Likewise, process P4 reads the value b, which is the most recent value that was returned by its preceding read by process P3. However, Fig. 2b shows a store that is not linearizable consistent since P3 reads an older value than its preceding read by process P2.

Versioning: Systems that process large quantities of data must often process the data in parallel. A version-based concurrency control algorithm can be an effective solution to maximize the number of operations performed in parallel. In this way, clients do not need to manage the synchronization.

The *Coverability* property, presented in [16], is an extension of linearizability which is more suited for versioned objects, like files. In particular, coverability allows multiple operations to modify the same version of an object concurrently, leading to a set of different versions. Besides the linearizability guarantees, it has the additional guarantee that object writes succeed when associating the written value with the "current" version of the object. On the other hand, a write operation becomes a read operation and returns the latest version and the associated value of the object.

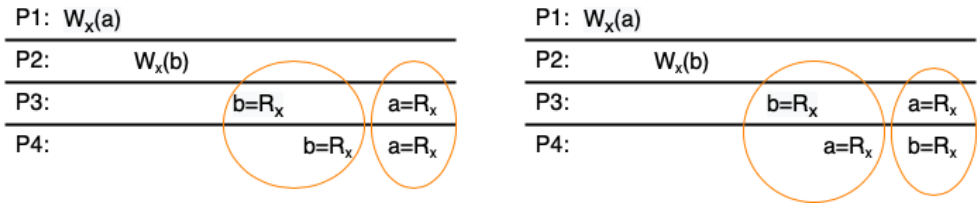
3.3 Sequential Consistency

The sequential consistency [17, 3], which is weaker than linearizability, represents a relaxation of the rules. A shared-memory system supports sequential consistency if all processes see the same order of memory access operations as all the other. That is, if one process sees one of the orderings of some operations, another process must see the same ordering; otherwise the memory is not sequentially consistent.

Fig. 3a depicts a sequentially consistent store, since processes P3 and P4 read the values written by processes P1 and P2 in the same order. However, Fig. 3b shows a store that is not sequentially consistent, since P3 and P4 disagree on the order of the write operations.

3.4 Causal Consistency

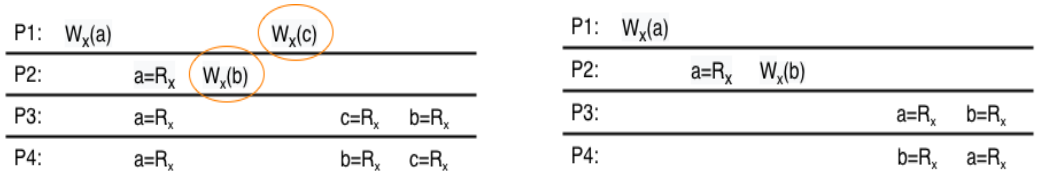
The causal consistency [3] relaxes the requirement of the sequential consistency by categorizing events into those causally related and those that are not. In the causal consistency model, only causally related write operations need to be seen in the same order by all processes. The



(a) (b)

Figure 3: Sequential Consistency Model

concurrent write operations may be seen in different orders by each process. Two writes are *causally related* if one write to a variable is dependent on a previous write to any variable if the processor doing the second write has just read the first write. On the other hand, if two operations write simultaneously to two different data items, these are not causally related, they are said to be concurrent.



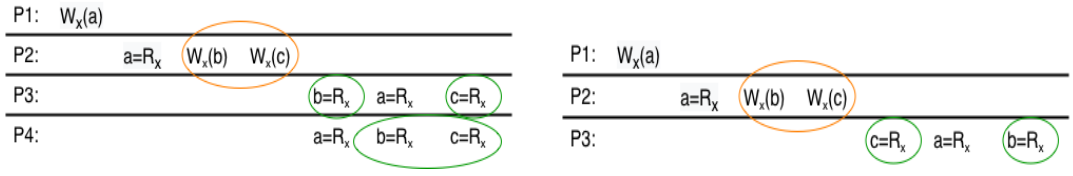
(a) (b)

Figure 4: Causal Consistency Model

The sequence in Fig. 4a is allowed with a causally consistent store, but not with a strictly or sequentially consistent store. Note that operations $W_x(b)$ and $W_x(c)$ by processes P2 and P1 write different values to x, and since they are concurrent, users can see different orders for them (there is no order guarantee). However, the data store in Fig. 4b does not provide causal consistency. $W_x(a)$ and $W_x(b)$ are causally related since P2, which executes $W_x(b)$ after $W_x(a)$ of P1, reads the value first. Thus, processes P3 and P4 have to see these writes in the same order.

3.5 FIFO Consistency

The FIFO consistency [3] (also known as Pipeline RAM consistency or PRAM consistency) relaxes the causal consistency model by maintaining ordering only based on the operations of the same process. In particular, writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes. As a result, this model does not guarantee the order in which different processes see writes generated by different processes.



(a) (b)

Figure 5: FIFO Consistency Model

Fig. 5a depicts a valid sequence of events for FIFO consistency. The only requirement in this example is that the writes of process P2 are seen by P3 and P4 in the correct order. However, the data store in Fig. 5b does not provide fifo consistency since the writes of process P2 are seen by P3 in incorrect order.

3.6 Relaxed Consistency

Relaxed Consistency [18] was introduced as an alternative to sequential consistency promising better performance by providing relaxations to the following characteristics:

1. *Program Ordering* requirement specifies whether they relax, the order from a write to a following read between two writes, and from a read to a following read or write. For example, the system may allow a processor to insert its write operation on a write buffer and proceed without waiting for this operation to complete. Subsequent, reads may be allowed to bypass any previous writes in the write buffer for faster completion.
2. *Write Atomicity* requirement, that is, whether they allow a read to return the value of another processor's write before all cached copies of the accessed location receive the invalidation or update messages generated by the write; in other words, before the write is made visible to all other processors. This relaxation only applies to cache-based systems.

Sometimes these two are combined together allowing a process to read the value of its own previous write before the write is made visible to other processes. Fig 6 depicts all relaxation requirements discussed above. Because of these relaxations, several other consistency models were implemented in both academic and commercial settings, varying on the degree of relaxations they provide. Between others, we have the Release Consistency, discussed in Section 3.11, and the GFS's (Google File System) relaxed model, discussed in Section 5.1.

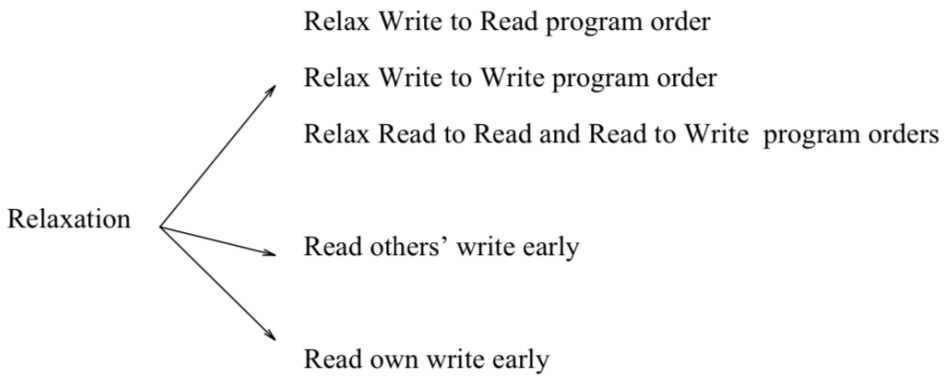


Figure 6: Relaxations allowed by memory models [18].

3.7 Bounded Staleness Consistency

Bounded Staleness Consistency [19], as its name implies, can tolerate staleness data, but up to a point. In particular, each read sees a subset of previous writes, where the level of stale of this subset is defined in terms of time. For example, some definitions, like the one in [19], define those writes to be completed some time interval before the read was issued (no stale data older than a specified time). Other definitions are based on the number of versions of the object (no stale data that's been updated more than a specified number of times).

Fig. 7 illustrates the bounded staleness consistency. After the data is written by process P1, process P2 reads the written value based on the configured time.

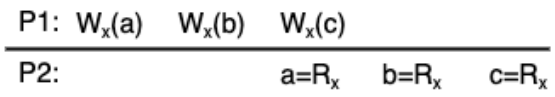
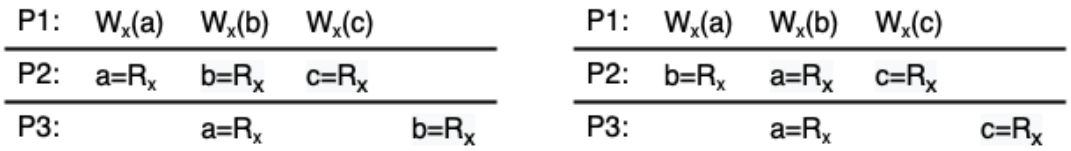


Figure 7: Bounded Staleness Consistency Model

3.8 Consistent Prefix Consistency

Consistent Prefix Consistency [20] is similar to bounded staleness but without the time guarantee. Thus, it can guarantee that readers see a consistent and ordered sequence of writes which may not be current. However, this model ensures that the user never sees a write out-of-order.



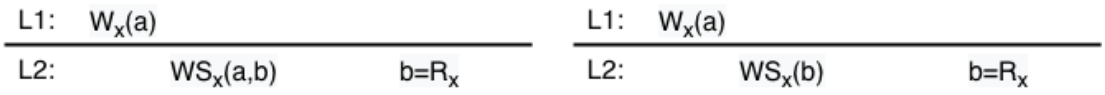
(a) (b)

Fig. 8a provides Consistent Prefix consistency. The data is written by P1 in the order A, B, C, so the processes P2 and P3 correctly read A,B,C and A,B. However, Fig. **??** does not provide Consistent Prefix consistency since the P2 and P3 read out-of-order entries, B,A,C and A,C respectively.

3.9 Eventual Consistency

Eventual Consistency [21, 22] guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. More specifically, updates may be seen by each process in arbitrary order, and processes may diverge for an unbounded amount of time and only eventually come to agree. Note that eventual consistency does not restrict the order in which the writes must be executed. Several systems implement eventual consistency models, including among others the well-known Domain Name System (DNS), git, iPhone sync, Dropbox and Amazon's Dynamo. The eventual consistency model has a number of variations.

- **Read-your-writes Consistency:** If a process P updates some value, a subsequent read by that process will always access the updated value and never see an older value. This is a special case of the causal consistency model.

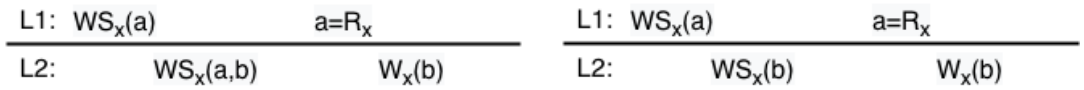


(a) (b)

Figure 9: Read-your-writes Consistency Model

The data store in Fig. 9a provides Read-your-writes consistency, since a write is completed before a successive read. However, the data store in Fig. 9b does not provide Read-your-writes consistency since $WS_x(b)$ does not include $W_x(a)$, and so the effect of the previous write at local copy L1 has not yet been propagated to L2. (Where WS is the write set, i.e., sequence of write operations of the object at the local copy.)

- **Session Consistency:** This is a practical approach of the previous model, which ensures read-your-writes consistency during a session. A session is a group of process interactions with the system that take place within a given time frame. As long as the session continues, the system guarantees read-your-writes consistency, remembering all writes have been done during the session. If the session terminates, a new session must be created and there is no guarantee that the system will remember the writes from the previous session.
- **Monotonic Read Consistency:** If a process P reads some value, a subsequent read by that process will always access this value or a more recent one.

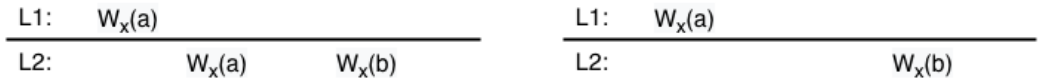


(a) (b)

Figure 10: Monotonic Read Consistency Model

The data store in Fig. 10a provides Monotonic read consistency. However, the data store in Fig. 10b does not provide Monotonic read consistency since $WS_x(b)$ does not include $W_x(a)$.

- **Monotonic Write Consistency:** The system guarantees to serialize the writes by the same process. More specifically, the write operation by process P on a value is completed before any subsequent write by that process.

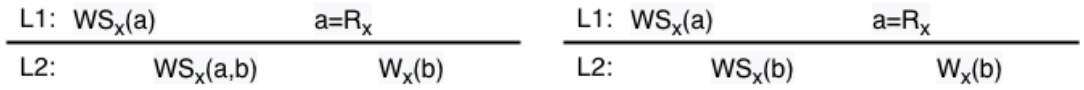


(a) (b)

Figure 11: Monotonic Write Consistency Model

The data store in Fig. 11a provides Monotonic write consistency, since the previous write operation at L1 has already been propagated to L2. However, the data store in Fig. 11b does not provide Monotonic write consistency, since a has not been propagated to L2.

- **Writes Follow Reads Consistency:** If a client reads some value, a subsequent write will operate on the same value or a more recent one.



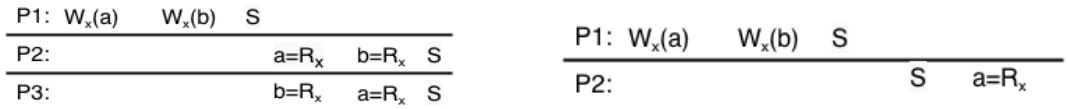
(a) (b)

Figure 12: Writes Follow Reads Consistency Model

The data store in Fig. 12a provides Writes follow Reads consistency. However, the data store in Fig. 12b does not provide Writes follow Reads consistency, since a did not appear in the write set at L2, where P later performs $W_x(b)$.

3.10 Weak Consistency

There are applications that do not need to see all writes, even from the same process, in the same order. This leads to weak consistency [3] that introduces the notion of a synchronization variable, which updates all the data of the shared memory. In contrast to the previous consistency models, this model enforces consistency on a group of operations, as opposed to individual reads and writes. The synchronization variable is used to delimit those groups of operations. The synchronization accesses are sequentially consistent. Before a synchronization access can be performed, all previous writes must be completed. As a result, the shared memory will only be consistent after a synchronization operation.



(a) (b)

Figure 13: Weak Consistency Model

Fig. 13a depicts a valid sequence of events for weak consistency. Processes P2 and P3 have not synchronised before executing the read operations, thus there is no guarantee about what order they see. However, Fig. 13b shows an invalid sequence of events for weak consistency since P2 performs a synchronization before executing the read, so P2 should see all updates.

3.11 Release Consistency

The release consistency [3] is a further relaxation of weak consistency without a significant loss of coherence. The release consistency model uses two types of synchronization variables: acquire and release operations. The acquire operation is called before entering a critical section and the release is called upon exiting. Critical Section refers to a section of code that attempts to access or modify the value of the variables in a shared resource. All pending acquire operations must be completed before a read or write operation is performed. Also, all pending acquire operations must be done before a release is done. These shared data kept consistent are called protected. However, the acquire and release do not have to apply to all data in a storage space.

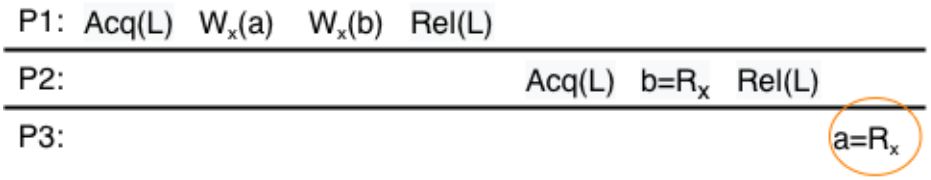


Figure 14: Release Consistency Model

Fig. 14 depicts a valid sequence for release consistency. Process P2 performs an acquire, so its read of "x" is consistent. However, process P3 has not performed an acquire, so there is no guarantee that the read of "x" is consistent.

3.12 Entry Consistency

Like the release consistency model [3], it also uses the acquire and release operations to enter and exit from a critical section, respectively. However, under entry consistency, every shared

variable is assigned a synchronization variable like a lock. This means that when an acquire is done on a synchronization variable, only the operations related to that synchronization variable need to be completed. This allows more than one critical section to be executed at a time, if the critical sections involve different shared variables.

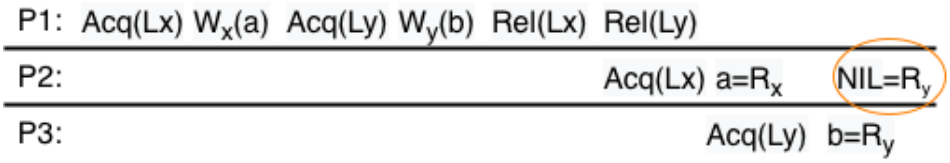


Figure 15: Entry Consistency Model

Fig. 15 depicts a valid sequence for entry consistency. In this example, locks are associated with each data item. However, the read of P2 on "y" returns NIL as no locks have been requested.

3.13 Models Comparison

Table 1 compares the most common consistency models in terms of the level of consistency, performance, and concurrency. In addition, Fig. 16 illustrates these popular models of distributed consistency, ordered by strength; smaller regions admit fewer executions.

Table 1: Consistency Models Comparison

Name	Consistency	Performance	Concurrency
Strict	highest	very low	very low
Atomic	very high	high	high
Bounded Staleness	high	high	high
Sequential	high	low	high
Causal	moderate	high	high
FIFO	moderate	high	high
Consistent Prefix	moderate	high	high
Relaxed	low	very high	high
Eventual	low	very high	high
Weak	very low	very high	high
Release	very low	very high	high
Entry	very low	very high	high

Strict Consistency is the ideal model that provides the highest consistency. However, its implementation in a distributed system is impossible since it relies on absolute global time, which significantly reduces the concurrency it can be achieved.

Linearizability/Atomic and Sequential Consistencies are easier and better than Strict since time does not play a role. Moreover, according to [17], supporting Linearizability provides better performance since it can be equally or more cost effective and provides faster response times than supporting Sequential consistency. The Bounded Staleness model has better performance than Linearizability however the availability is still low.

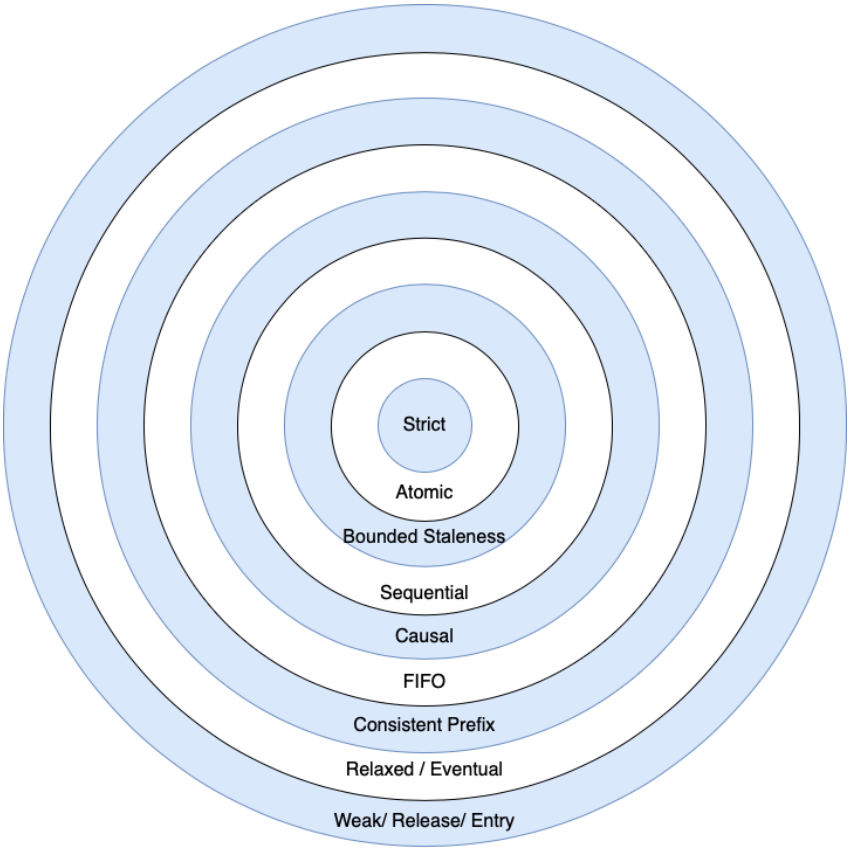


Figure 16: Consistency Hierarchy

Causal and FIFO consistency do not maintain global ordering of operations. Causal consistency makes a distinction between events potentially causally related and those not. The disadvantage of this technique is that it requires keeping track of which processes have seen which writes. FIFO consistency is easy to implement, but not all applications require seeing all writes.

The Consistent Prefix model provides high availability and very low latency. For both Relaxed and Eventual consistency models, for the reasons presented in the previous sections, even though they provide fewer consistency guarantees, they offer better performance and concurrency level. Relaxed consistency introduces relaxations based on program order and write atomicity, which enable many of the optimizations that are constrained by sequential consistency. Eventual consistency makes sure that data of replica servers gets consistent eventually. But, the time taken by the nodes of the database to get consistent is unspecified.

The weak consistency models, including Weak, Release and Entry consistency use synchronization variables. Thus, synchronization occurs only when shared data is locked and unlocked. However, the weaker the consistency model, the easier it is to build a scalable solution.

4 Distributed Shared Memory Emulations

Asynchrony [7, 23] is an important challenge during the design of a distributed storage algorithm. Clients access the storage over connection with unpredictable delays, such as the Internet. Therefore, it is impossible for clients to realize if a process is just slow or if it may experience a failure. We assume a memory-passing system consisting of three distinct sets of processes: a

set $\mathcal{W}$ of writers, a set $\mathcal{R}$ of readers, and a set $\mathcal{S}$ of servers. Let $\mathcal{C} = \mathcal{W} \cup \mathcal{R}$ be the set of clients.

- **writer** $w \in \mathcal{W} \subseteq \mathcal{C}$ **:** a client that dispatches update requests to servers.
- **reader** $r \in \mathcal{R} \subseteq \mathcal{C}$**:** a client that dispatches read requests to servers
- **server** $s \in \mathcal{S}$**:** listens for reader and writer requests and is responsible for maintaining the object replicas according to the underlying shared memory they implement.

A shared register [24, 25] is a readable and writable object which is used to store the value on which the read/write operations are being performed. Let R denote a register. The read operation, denoted by *read*, takes no arguments and returns the value of the register R. A write operation, denoted by $write(v)$, takes an argument v, changes the value of R to v and returns *ack*. We assume that the registers are initialized to some default value. A register is characterized by the number of writers and readers that participate in the system. The Single Writer, Multiple Readers model, denoted as SWMR, has $|\mathcal{W}| = 1$ and $|\mathcal{R}| \geq 1$. While the Multiple Writers, Multiple Readers model, denoted as MWMR, has $|\mathcal{W}| > 1$ and $|\mathcal{R}| \geq 1$. Registers can also be classified according to the consistency condition they provide when read operations invoked in the presence of concurrent write operations [26, 7]. Lamport [24, 25] defined three consistency semantics for registers: *safe* registers, *regular* registers and *atomic* registers. *Safe* registers ensure that a read operation that is not concurrent with any write operation returns the value written by the last write operation. However, a read operation that is concurrent with a write operation returns an arbitrary value of the possible values of the register. *Regular* registers provide a stronger guarantee. They ensure that a read operation that is concurrent with a write operation returns either the value of the last preceding or the new write. *Atomicity/linearizability* is the strongest consistency semantic that provides the illusion that the storage is accessed sequentially, that is, the read operations return the same values as if the operations had been performed in that order. This is achieved using some way of total order on all its operations. This is essentially the linearizability consistency model of Section 3.2. In addition, the register is also characterized by the type of failures that it can tolerate, e.g., *crash* failures, *arbitrary* failures. A crash failure occurs when a node halts once, and then stops responding completely, and becomes unresponsive (aka, it crashes). In synchronous systems, it is possible to detect a crash failure (using heartbeat or timeout). Fischer, Lynch and Paterson [5] presented the FLP Impossibility Theorem which states that it is *impossible* in an asynchronous system for a set of nodes to reach agreement if at least one node fails even by crashing; asynchrony prevents distinguishing between process crashes and delays. Thus, in asynchronous systems, the detection of crash failures is never accurate, since it is not possible to distinguish between a process that has crashed and a process that is running slow. Byzantine failures [6] are more complicated because the faulty unit can continue to be active and return incorrect/corrupted values, or not respond at all. In particular, a Byzantine node may arbitrarily deviate from its specified protocol. This can be because of faults in the system (hardware, software bugs, etc.) or because of malicious intent. An example of malicious behaviour is when a process executes a different program instead of the specified one. Byzantine faults are obviously the most difficult kind of failure to deal with because the behavior is non-deterministic and the node might appear healthy to the rest of the network.

The environment of a distributed system can be either *static* or *dynamic*. In a static environment, the set of S remains fixed, even if servers fail, and each server in the set maintains a replica of the object. In dynamic environments the set of participating servers may dynamically change over time due to servers removal or addition, and the object is replicated at this subset of server participants. To deal with the dynamic environment, reconfigurable algorithms are

introduced. A reconfigurable register [27, 28] relies on a configuration object which includes the set of servers and some additional information that is needed. Configurations are changed by a procedure called *reconfiguration*. A reconfiguration can generally add or remove servers. Thus, a reconfigurable algorithm performs three main operations: read, write, and introduce new configurations.

There are many interesting works which implements algorithms for shared memory emulation. These algorithms are designed to emulate a shared memory over an asynchronous, distributed message-passing system, i.e., over a set of communicating servers. In this work, we focus only on atomic (linearizable) shared memory emulation algorithms. In this section, we present some of the most popular ones. We divide them into two main categories: Crash-tolerant DSMs and Byzantine-tolerant DSMs. We further separated the crash-tolerant algorithms into static algorithms, including some that support "fast reads", and dynamic algorithms that support reconfiguration both with and without consensus.

Due to asynchronous environment, the read and write operations have to be ordered. In SWMR environment, read and write operations are ordered using logical timestamps associated with each written value. These timestamps totally order write operations, and therefore determine the values that read operations return. The timestamp is an integer number that lets the system to distinguish old from new values. However, in MWMR environment, when a writer performs a write operation it associates the value with a tag $<ts, id>$, where ts is a logical timestamp, and id is the writer's unique id that distinguishes the current write operation from all others. Tags are ordered lexicographically, first by the ts, and then by the id, in establishing an order on the operations.

4.1 Crash-tolerant DSMs

Table 2 presents a comparison of the main characteristics of crash-tolerant DSMs that we will discuss in this subsection. Below we present several solutions developed for static and reconfigurable atomic memory algorithms.

4.1.1 Static Server Participation

We now survey several algorithms that implement atomic shared memory in the asynchronous, message-passing, crash-prone, static setting. We present several solutions developed for both the SWMR and MWMR settings.

4.1.1.1 **Algorithm** ABD

Attiya, Bar-Noy and Dolev [42] present the first fault-tolerant emulation of atomic shared memory in an asynchronous message passing system, also known as *ABD* emulation. It implements SWMR registers in an asynchronous network, provided that at least a majority of the servers do not crash.

As seen in Fig. 17, the writer completes write operations in a single round-trip communication, round for short, by incrementing its local timestamp and propagating the value with its new timestamp to the servers, waiting for acknowledgments from a majority of them. The read operation, in Fig. 18, completes in two rounds: (i) it discovers the maximum timestamp-value pair that is known to a majority of the servers, (ii) it propagates the pair to the servers, in order to ensure that a majority of them have the latest value, hence preserving atomicity. This has led to the common belief that "atomic reads must write". Fig. 19 shows an example of what would happen if the second phase of read did not exist. All servers start with the value v and a timestamp (ts) $v0$. A slow writer gets to write the value $newV$ on only one server before two readers are performed. The first read gets replies from three servers, two of which have the

Table 2: Comparative table of Crash-tolerant DSMs.

Algo.	SWMR or MWMR	Data scalab.	Version.	Write Exch.	Read Exch.	Recon.	Repl. or EC
SWMR ABD [29]	MWMR	NO	NO	2	4	NO	Repl.
MWMR ABD [29]	MWMR	NO	NO	4	4	NO	Repl.
LDR [30]	MWMR	YES	NO	4	4	NO	Repl.
CoABD [16]	MWMR	NO	YES	4	4	NO	Repl.
Fast [31]	SWMR	NO	NO	2	2	NO	Repl.
SF [32]	SWMR	NO	NO	2	2 or 4	NO	Repl.
SLIQ [33]	SWMR	NO	NO	2	2 or 4	NO	Repl.
CwFr [34]	MWMR	NO	NO	4	2 or 4	NO	Repl.
SFW [35]	MWMR	NO	NO	2 or 4	2 or 4	NO	Repl.
OhSam [36]	SWMR	NO	NO	2	3	NO	Repl.
OhMam [36]	SWMR	NO	NO	4	3	NO	Repl.
ccFast [37]	SWMR	NO	NO	2	2	NO	Repl.
SWMR Erato [38]	SWMR	NO	NO	2	2 or 3	NO	Repl.
MWMR Erato [38]	MWMR	NO	NO	4	2 or 3	NO	Repl.
ccHybrid [37]	SWMR	NO	NO	2	2 or 4	NO	Repl.
OhFast [37]	SWMR	NO	NO	2	2 or 3	NO	Repl.
RAMBO [27]	MWMR	NO	NO	4	4	YES	Repl.
DynaStore [39]	MWMR	NO	NO	≥ 4	≥ 4	YES	Repl.
SM-Store [28]	MWMR	NO	NO	4	4	YES	Repl.
SpSn-Store [40]	MWMR	NO	NO	4	4	YES	Repl.
Ares [41]	MWMR	NO	NO	4	4	YES	EC

old value v and one of which has the new value $newV$. Thus, it returns the value $newV$. The second read, on the other hand, only receives responses from servers that keep the old value; thus, it returns the value v. This violates the atomicity since the second read returns an older value than the first one. Whereas if the first read executed the second phase, it would propagate the value to a majority.

The key to the correctness of the algorithm is to notice that each operation communicates with a majority of the servers: since $n > 2f$, where n is the total number of the participating servers and f is the number of the servers that may fail, it follows that $n - f > n/2$. Thus, there is at least a common server communicating with each pair of write and read operations, ensuring that the value of the latest non-overlapping write operation is forwarded to the later read operation. Regarding crashes, any number of readers may crash. If the writer crashes, then no further updates take place, unless another writer enters the system.

The ABD algorithm was extended by Lynch and Shvartsman [29], who present an emulation of MWMR atomic registers in message-passing systems. Also, they generalized majorities into *quorums*. A quorum is a subset of the set of servers, and each quorum has a common intersection with every other quorum in the system. In this work, both write and read operations perform 2 rounds, a query round to discover the maximum tag, and a propagation round to inform the

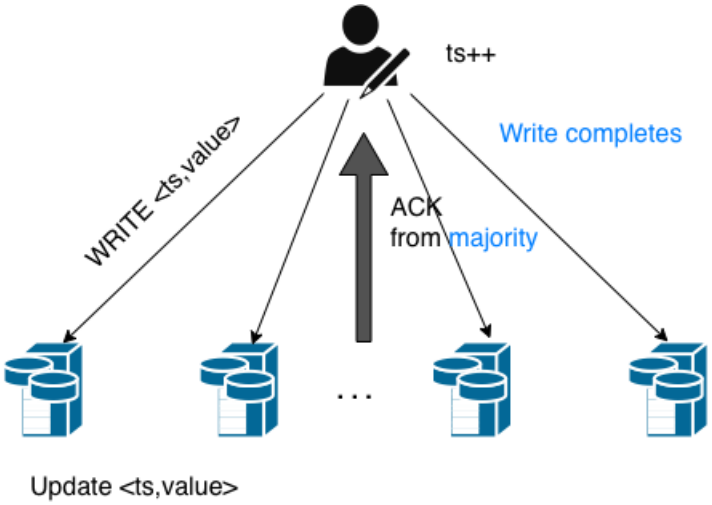


Figure 17: SWMR ABD - write operation

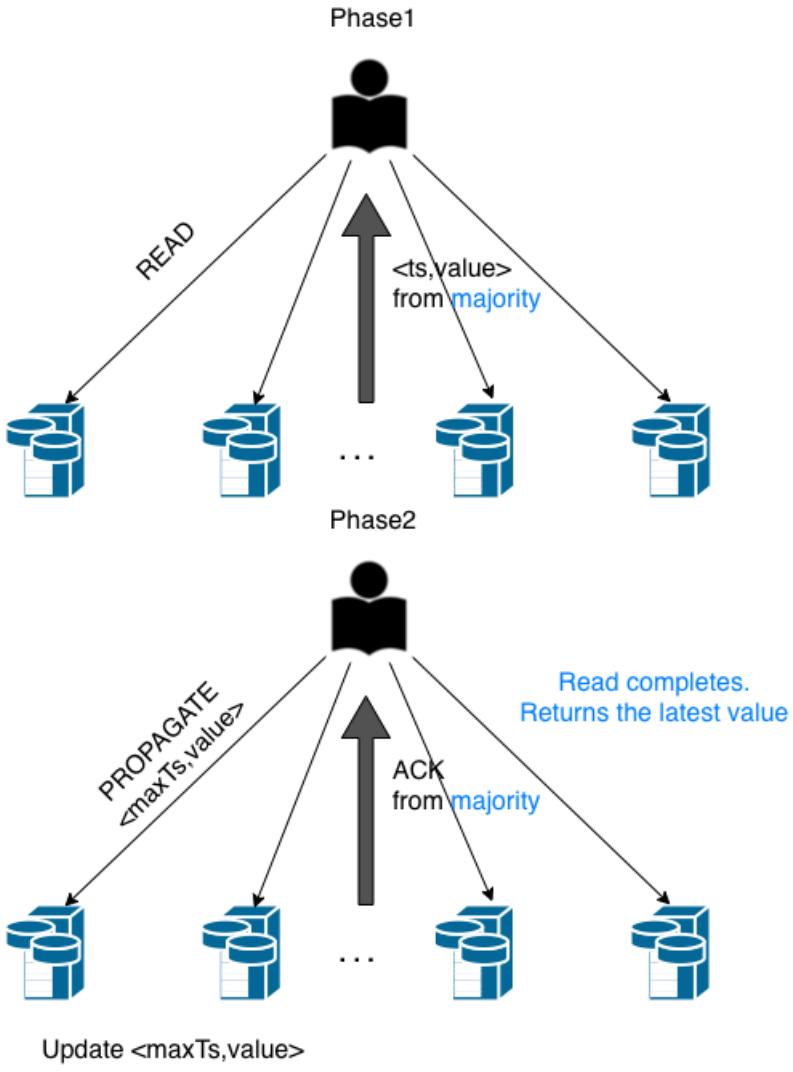


Figure 18: SWMR ABD - read operation

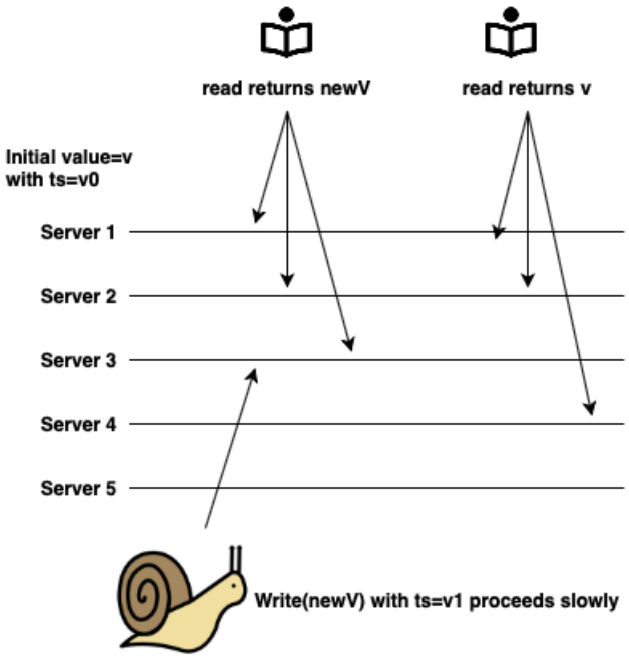


Figure 19: Example where atomicity is violated if the second phase of read did not exist.

servers of its local tag. The read operation is identical to the four-exchange protocol in SWMR ABD, the only difference being that tags are used instead of timestamps. The write operation with 2 round-trips is shown in Fig. 20. The only difference between a write and a read lies in the second round, where the write operation increments the maximum tag, while the read operation propagates this maximum tag. The first two exchanges ensure that the writer produces a tag that is higher than that of any preceding write.

In shared objects, a write operation should extend the latest written version of the object, and not overwrite any new value. In this respect, the notion of *coverability* was introduced in [16], which is a consistency guarantee that extends linearizability and concerns versioned objects (as described in Section 3.2). In that same work an implementation of a coverable (versioned) object was presented, which we call CoABD. Read operations are identical to those of MWMR ABD, with the difference that they return both the version and the value of the object. Write operations, on the other hand, attempt to write a "versioned" value on the object. If the reported version is older than the latest version of the object, then the write does not take effect and is converted into a read operation. This way, writes are prevented from overwriting a newer version of the object.

4.1.1.2 **Algorithm** LDR

Fan and Lynch [30] proposed an extension of ABD, called Layered Data Replication (LDR), that can cope with large-size objects (e.g., files). It is called LDR, because it has two layers of servers. The key idea was to maintain copies of the data objects separately from their metadata by maintaining two different types of servers, replicas and directories. The replica servers store the whole data object, while the directories store metadata, including timestamps and the set of replica servers that have information about the most up-to-date copies. Essentially the clients with the directory servers run ABD-like emulations and clients contact the replica servers to retrieve the object or to update its content.

This algorithm deals with data replication by taking advantage of the fact that metadata

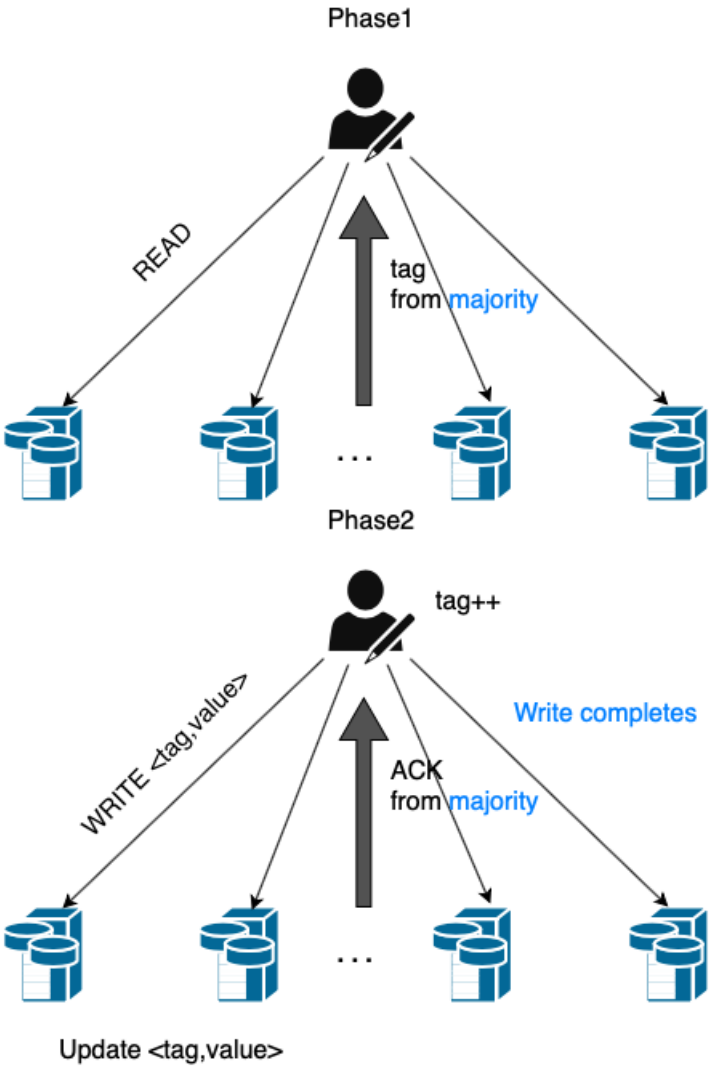


Figure 20: MWMR ABD - write operation

are lighter than replicated actual data. Thus, it performs more operations on the metadata than on the actual data (files).

The implementation consists of m clients, a set R of Replica Servers and a set D of Directory Servers. The state of a client is described by three variables: (1) variable *phase* that takes a value according to the operation that takes place at a time, and (2) variable *utd* stores the set of replicas that the client thinks are the most up-to-date and variable *mid* (initially 0), which is a message counter.

A replica server, or replica for short, has one state variable, a triple consisting of: (1) the actual value of the object that it stores, (2) a tag that is associated with this value, and (3) information about on whether this value is considered "secured" or not; this parameter is used when the replicas perform garbage collection. Finally, a directory server, or directory for short, which is responsible to respond to client requests, has a state variable that stores the set of replicas having the latest value of a data object, and another that is a tag associated with that value of the data object.

In the following, we describe in an abstract way how read and write operations are performed in LDR.

For a read operation, a client first asks the directories to find the set of up-to-date replicas, and then reads the data from one of them. As shown in Fig. 21, a read operation consists of four phases:

1. read-directories-read (rdr):The client reads (utd, tag) from a quorum of directories to find the most up-to-date replicas and updates its local parameters based on the responses received.

2. read-directories-write (rdw): The client updates a quorum of directories sending (utd, tag) to a quorum of directories.

3. read-replicas-read (rrr): The client reads the value of the object which is associated with the latest found tag, from a replica in *utd*.

4. read-ok (rok): The client returns the value it read in rrr.

For a write operation, a client first writes its data to a set of replicas, and then informs the directories that these replicas are now up-to-date. As shown in Fig. 22, a write operation consists of four phases:

1. write-directories-read (wdr): The client reads (utd, tag) from a quorum of directories, and updates its local information based on the responses received setting its tag to be higher than the largest tag it read.

2. write-replicas-write (wrw): The client writes (v, tag) to a set of *acc* replicas, where $|acc| \geq f + 1$ and f is the maximum number of allowed replica fails and must be less than $|R|$. For example, it can send to $2f + 1$ replicas and wait $f + 1$ acknowledgements.

3. write-directories-write (wdw): The client writes (acc, tag) to a quorum of directories, to inform them for the most up-to-date set of replicas and the newest tag.

4. write-ok (wok): Then, it sends each replica a secure message to tell them that its write is finished, so that replicas can garbage collect old values of the object.

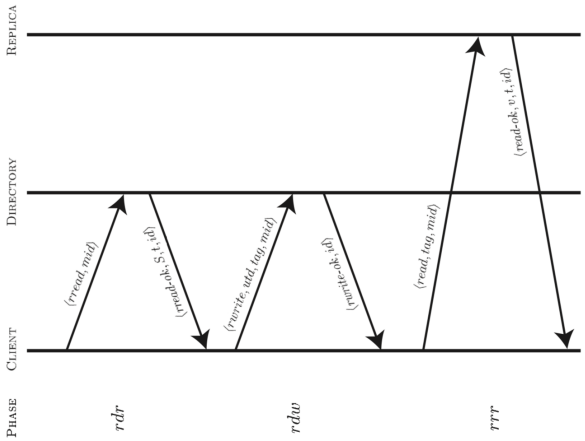


Figure 21: LDR read operation [30].

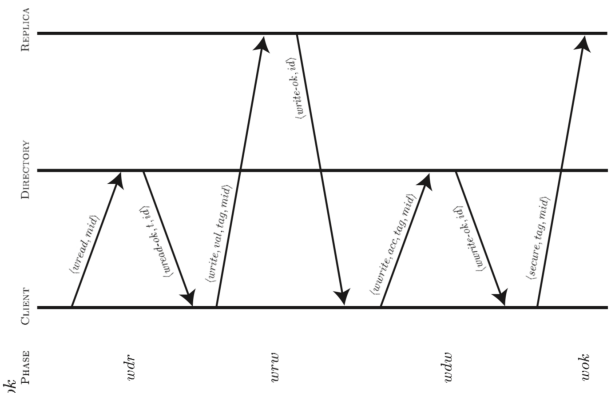


Figure 22: LDR write operation [30].

4.1.1.3 Algorithm Fast

Dutta et al. [31] presented an SWMR algorithm (we refer to it as Fast) in which all read and write operations required just a single communication round before completing, under a certain constraint. In particular, it guarantees correctness only when the number of readers is constrained with respect to the number of servers and in inverse proportion to the number of crashes $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$. Fast uses $< timestamp, value >$ pairs as in SWMR ABD to impose an order on the write operations. The write operation has one round: the writer increments its local timestamp, and sends the new timestamp with the value to be written to the majority of the servers. To avoid the second round in read operations, Fast uses two mechanisms: (i) a recording mechanism at the servers, and (ii) a predicate that uses the server records at the readers. Essentially each server records all the processes that witness its local timestamp, in a set called *seen*. This set of processes is reset whenever the server learns a new timestamp. The predicate at the readers checks the following: The reader collects timestamps from $\mathcal{S} - f$ servers, and selects the highest timestamp. Then the reader checks if this timestamp has been *seen* by a sufficient number of servers and readers. If the predicate holds, the reader returns the value associated with the maximum timestamp. Otherwise, it returns the value associated with the previous timestamp.

4.1.1.4 Algorithm SF

Algorithm SF [32] ("semifast") introduces a similar predicate in the readers to the one of algorithm Fast to determine whether it is safe for a read operation to terminate after one round and allows an arbitrary number of readers. Notice that the predicate of Fast examines which processes witnessed the latest timestamp as it examines the intersection of the *seen* sets. The main difference between SF and Fast is that the predicate does not examine all the possible subsets of readers, grouping the readers into logical sets, called *virtual groups*. With this modification, the algorithm does not depend on the number of concurrent readers in the system, but it is based on forming groups of processes where each group is given a unique virtual identifier.

In particular, each server records the virtual group of each reader requesting its local value, and attaches to each reply the set of virtual group identifiers it has. If the same server is contacted by two readers from the same virtual group, the set will only contain one instance of the group identifier. Upon receiving a reply, the reader applies the predicate to the set of virtual nodes. If "enough" processes witnessed the $maxTS$, then the reader performs the second phase before completing. Otherwise, the read is fast. Since each virtual group may grow arbitrarily,

SF supports unbounded number of readers.

The work of Kentros et al. [43] presents performance results obtained by simulating algorithm SF using the NS-2 network simulator. The goal of this work was to quantify the number of occurrences of slow reads in executions under practical settings. The authors examined how the algorithm behaved in environments with low and high contention (i.e., the volume of concurrent operations). The findings showed that in low contention, $O(log\mathcal{R})$ slow read operations can be sufficient for each write operation. Furthermore, if contention is high, the system may reach a point where ($log\mathcal{R}$)reads can be completed, allowing for up to $\mathcal{R}$ slow reads.

4.1.1.5 Algorithm SLIQ

SLIQ [33](Semifast Like Implementation for Quorum systems) is a SWMR algorithm that allows fast reads after the completion of a write operation. To this respect, the notion of *quorum views* was introduced. Quorum views are client-side decision tools (their use is explained below). A write operation takes one communication round to complete, the propagation of a write message to all the servers. The writer waits for replies from a full quorum, and then it increments its timestamp and the operation completes. On the other hand, a read operation terminates in either one or two communication rounds (i.e., two or four communication exchanges). A read operation also performs a propagation phase. Once the reader receives replies from a full quorum, it examines the distribution of the maximum timestamp ($maxTS$) within that quorum of replies. There are three cases of Quorum Views which a reader can observe after the first round: (i) all the members of a quorum have the $maxTS$, (ii) a subset of members of each intersection maintains an older timestamp, and (iii) some intersection contains $maxTS$ in all of its members. In either case (i) or (ii), the read completes and returns the value associated with the discovered $maxTS$ or $maxTS - 1$ respectively. In the case (iii), the reader continues into the second round, propagating the $maxTS$ and its associated value to a full quorum and waits replies from a full quorum. After that, the read operation completes, returning $maxTS$ and its associated value.

Fig. 23(a) depicts case (1) in which a full quorum reports the same $maxTS$, implying that the write operation may be completed. Fig. 23(b) depicts case (2) in which a subset of members of each intersection retains an older timestamp, implying that the write operation that propagates $maxTS$ is not yet complete. Finally, Figs. 23(c) and 23(d) show two different examples of case (3). In the former, a write operation that is not completed propagates the $maxTS$ in the dark nodes, while in the latter, it is completed by getting responses from the Qz.

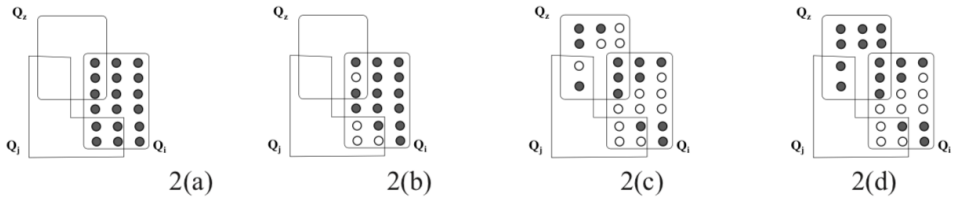


Figure 23: The possible completion of the write operation [33].

4.1.1.6 Algorithm CwFr

CwFr is an MWMR algorithm [34] that allows fast read operations by adopting the general idea of Quorum Views from Algorithm SLIQ. Multiple writers in the system prohibit the assumption that when the latest write operation is non-completeness, any previous write operation is already completed. Multiple writers may write different values (or tags) concurrently. Write operations take two rounds, the query and propagate like in ABD. On the other hand, a read operation

terminates in either one or two communication rounds. The read operation performs an iterative approach, in which analyzes the distribution of the maximum tag within a quorum of replies from servers, to determine the latest potentially completed write operation. There are three cases of Quorum Views which a reader can observe after the first round: (*i*) all the members of a quorum have the $maxTag$, (*ii*) a subset of members of each intersection maintains an older tag, and (*iii*) some intersection contains $maxTag$ in all of its members. In case (*i*), the read completes and the value associated with the discovered $maxTag$ is returned. In case (*iii*), the reader continues into the second round, propagating the $maxTag$ and its associated value. However, in case (*ii*), the read operation performs an iterative approach, removing the servers that replied with the $maxTag$. After each iteration, the reader determines the next $maxTag$ in the remaining server set, and then re-examines the quorum views in the next iteration until case (*i*) or (*iii*) is observed.

4.1.1.7 Algorithm SFW

SFW [35] enables fast read and write operations and it is the first algorithm that allows fast write operations in the MWMR setting. Like the previous algorithms, SFW also uses tags to ensure ordering in write operations. However, to achieve fast writes, it introduces a new technique, called *Server Side Ordering* (SSO). Using SSO, the writer may avoid the query phase of a write operation, completing the write in one communication round. This technique transfers from writers to servers the responsibility of incrementing the tag associated with every value written. In particular, during the first round, the writer sends its local tag tag_w and the new value to the servers. Once each server receives this message, it finds the maximum tag between its local and received (tag_w) tags, icrements the maximum one, and replies back to the writer with the generated tag. Then, the writer uses a predicate to decide if it collects the same tag from a sufficient number of servers. If the predicate holds, the write operation is completed in a single round.

4.1.1.8 Algorithms OhSam and OhMam

OhSam [36] is a SWMR algorithm, where each read operation takes one and a half round-trips to complete. OhSam is the first algorithm to consider server-to-server communication, as opposed to previous works that only considered client-server communication round-trips. Writes take just one round/2 communication exchanges (similarly to ABD) and reads always take one and a half round/3 communication exchanges to complete. The main idea of the algorithm is to allow servers to exchange information about the operations, before replying to the invoking client. The three communication exchanges of Read operations are: (*i*) the reader sends message to servers, (*ii*) each server that receives the request relays the request to all other servers, and (*iii*) once a server receives the relay for a particular read from a majority of servers, it replies to the reader. The read completes once it collects a majority of these replies. A key idea of the algorithm is that the reader returns the value that is associated with the minimum timestamp. Due to asynchrony it is possible for messages from the third exchange to arrive at the reader before messages from the second exchange.

The authors in [36] also proposed an extension of OhSam and present a MWMR algorithm, called OhMam. Again, each read operation takes one and a half round-trips to complete. The read operation is almost identical with OhSam's read operation. The only difference is that OhMam uses tags instead of timestamps. However, the write protocol is similar to the one that MRMW ABD uses and completes in four communication exchanges.

4.1.1.9 Algorithms ccFast and ccHybrid

The authors in [37] modify the Fast algorithm to make it even faster. They present a new

algorithm, called ccFast, that (*i*) reduces the size of messages and (*ii*) reduces the computation time of the predicate. It has the same constraint on the number of readers as Fast has. It involves only two communication exchanges. The authors modified the predicate of Fast algorithm: the idea of the new predicate is to examine the replies received in the first communication round of a read operation and determine how many (instead of which in Fast) processes witnessed the maximum timestamp among those replies. With this modification, the predicate takes polynomial time to decide the value to be returned, and it reduces the size of each message sent by the nodes.

The authors in [37] also proposed algorithm ccHybrid which aims to allow an unbounded number of readers to participate in the service while allowing operations to complete in either two or four communication exchanges. In particular, ccHybrid combines ideas from algorithms ccFast and ABD: (*i*) it exploits timestamp-value pairs to order the write operations, (*ii*) it uses a predicate proposed by ccFast to determine the value returned by a fast read and (*iii*) it propagates the maximum timestamp-value pair to a majority of servers during a slow read as in ABD.

4.1.1.10 Algorithm OhFast

OhFast [37] is a SWMR algorithm, which pushes the responsibility of deciding whether a slow read operation is necessary or not to the servers. The algorithm performs either the fast read operations by ccFast or the one-and-a-half-rounds operations of OhSam. When servers determine that a slow read is necessary (i.e., the number of processes that requested their timestamp is large), they perform a relay phase to inform other servers before replaying to the reader. Thus, some of the servers may reply directly to the reader, whereas some others may perform a relay phase first. However, a read operation may terminate before receiving a reply from a relaying server. When a server that relays a timestamp gets $|S| - f$, where f is the number of server failures, relays from the other servers, it marks the timestamp as secured, and sends a reply to the reader. At the first round, a reader sends read messages to all the servers and waits $|S| - f$ replies. Once it receives these replies, the reader finds the messages with the highest timestamp ($maxTS$). If some of these messages indicate this $maxTS$ as secured, then the read operation completed and returns the value associated with this timestamp. Otherwise, the reader evaluates the predicate of ccFast on the replies to decide on which value to return. If the predicate holds, then the reader returns the value associated with $maxTS$, otherwise the value associated with $maxTS - 1$.

4.1.1.11 Algorithm Erato

Erato [38] is a SWMR algorithm, which improves the three-exchange read protocol of OhSam and allows reads to terminate in either two or three exchanges. This is achieved by using quorum views of SLIQ algorithm. Like in OhSam, Erato has three exchanges, where the former is initiated by the reader, and the other two are conducted by the servers: (*i*) the reader sends the request to servers, (*ii*) each server that receives the request relays the request to all other servers and the reader. The reader terminates if it receives relay messages from a quorum and, based on the distribution of the timestamp it notices the quorum view 1 or 2 of CwFr. Otherwise, the read operation terminates after three communication exchanges.

The authors in the same paper [38] also proposed the MWMR version of Erato. In the case where there is a single writer in the system, Erato can be relied upon on the fact that if the reader knows that the write operation is not complete, then any previous write is complete. However, this is not possible in concurrent writes. Therefore, the MWMR Erato uses the algorithm OhMam with the iterative technique using quorum views of CwFr.

4.1.2 Reconfigurable Implementations

Reconfiguration (i.e., reconfig operation) is the process of changing the set of servers, i.e., inserting or removing servers in DSS. A main challenge when supporting reconfiguration is ensuring consistency when multiple users submit concurrent reconfiguration requests. Below we describe five well-known algorithms for reconfigurable atomic storage, with and without consensus. These algorithms ensure that the quorum configuration of the servers changes dynamically, without interrupting ongoing read and write operations. *Consensus* [44] is a form of agreement which clients use to agree on the next configuration, and then they transfer the state of the object to this new configuration. Consensus can be ideal in *partially synchronous* systems [45]. Partially synchronous systems lie between synchronous and asynchronous ones; there are bounds on transmission delay, clock drift, and processing time, but they are not known to the algorithm. However, in asynchronous systems, we have to resort to a weaker form of agreement, *Lattice Agreement* [44], in which clients choose potentially different but comparable configurations.

4.1.2.1 Algorithm RAMBO

RAMBO [27], which stands for "Reconfigurable Atomic Memory for Basic Objects", is the first algorithm to address the reconfiguration of atomic MWMR storage. It supports shared atomic read/write shared objects in a distributed asynchronous environment with crash-prone nodes joining and leaving the system dynamically. The core of RAMBO is the component that performs read and write operations using a variation of MWMR ABD (cf. Section 4.1.1.1). Also the algorithm supports a second mechanism called Recon, which is responsible for performing reconfigurations and uses consensus to decide on reconfigurations. Recall that a reconfiguration is the act of changing the quorum configuration, and each configuration includes the set of active nodes that replicate the object. The architecture of a reconfiguration process is composed of three components:

1. Joiner: This component is responsible to handle requests from new nodes to join the system.

2. Reader-Writer (RW): This component is responsible for executing the read and write operations initiated by clients, as well as for performing garbage collection of old quorum configurations. Both read and write operations consist two phases. The first one is the query, in which information is retrieved from one or more quorums. The second phase is the propagate where information is send to one or more quorums. In the case of a write operation, the information propagated is the new object's value, where in a read operation is the value that is being returned. As already mentioned, the Reader-Writer (RW) component is also responsible for the garbage-collection operation. It also consists of two phases. In the first "Query" phase, RW contacts a quorum from the old configuration (k) of the system to ensure that all processes in that quorum learn about configurations k and $k+1$ and also learn that all configurations smaller than k have been garbage-collected. In the second "propagation" phase, RW contacts a quorum of configuration $k+1$ propagating for any changes during the second phase. On the completion of the second phase, the old configuration is removed from the system.

3. Recon: This component is responsible for managing reconfiguration. When a client wants to reconfigure the system, it initiates a recon request to propose a new configuration. To select one of the proposed configurations, RAMBO uses a consensus protocol, such as Paxos [46]. The main requirement that RAMBO needs by the Recon component is that

the configurations must be seen in the same order from each client. In particular, once clients learn about the configuration in the k'th index, they must agree on what that is.

4.1.2.2 Algorithm DynaStore

DynaStore [39] is a dynamic atomic memory service for MWMR objects. It is the first algorithm that solves the atomic R/W storage problem in a dynamic setting without the use of consensus. Like RAMBO, DynaStore supports three kinds of operations: read, write, and reconfig. Again, the protocol of reads and writes is similar to ABD and both operations are completed in two rounds. A participant uses the reconfig operation to change its current configuration (also called view) and gives as input a set of changes. This set includes elements of the form $(+, i)$, indicating the addition of replica i, and $(-, j)$, indicating the removal of replica j.

For each configuration c, DynaStore uses a weak snapshot $nextConfig(c)$ to store the next configuration. This weak snapshot object supports update and scan operations on a given set of processes; a scan operation returns a set of updates and sees the "first update" made to the object. Processes update $nextConfig(c)$ to suggest the next configuration after c (concurrent updates are possible). Unlike RAMBO, which establishes a sequence of configurations, weak snapshots organise all views into a DAG (directed acyclic graph) of configurations. Vertices in the DAG correspond to views and edges correspond to configuration changes executed by clients.

Both read and write operations first execute read information on reconfiguration and then execute the read-phase (similar to Read phase of ABD), where they query a quorum of the processes for the latest value and timestamp. If a new view is discovered, the read-phase repeats with the new view. Then they execute a write-phase, in the last view found by the previous phase: a read operation writes the value and timestamp obtained in the query phase, whereas a write operation writes a higher timestamp and the new value. Then it reads the reconfiguration information. If a new view is discovered, the protocol goes back to the read-phase in the new view.

As mentioned above, a $reconfig(cng)$ operation suggests changes using elements of the form (+, i) and (-, j). A reconfig operation is similar to a read, with the only difference that initially the set of configuration changes is not empty. In the reconfiguration example of Fig. 24, suppose that the initial configuration (C_0) is A, B, C. A process a invokes $reconfig(+D)$, while another process b invokes $reconfig(-C)$. Thus, process a updates $nextConfig(C_0)$ to C_1. The process a scans $nextConfig(C_0)$ to check for concurrent updates. This scan returns C_1 which indicates that there are no concurrent updates. Thus C_1 is the next established configuration after C_0. Finally the state of process a is transferred along all paths in the DAG; process a reads from a majority of C_0 and a majority of C_1 and then writes the latest value found to a majority of C_1. Concurrently, process b updates $nextConfig(C_0)$ to C_2 and scans it. This scan returns C_1, C_2, implying that the update of process a was concurrent. Thus, process b updates $nextConfig(C_1)$ and $nextConfig(C_2)$ to C_3. Since no concurrent updates detected, C_3 is an established configuration. Finally, the state of process b is transferred: process b reads from a majority of each configuration on every path found from C_0 to C_3 and writes the latest value found to a majority of C_3.

4.1.2.3 Algorithm SM-Store

SM-Store (SmartMerge) [28] is very similar to DynaStore, but it uses a SmartMerge function that combines concurrently issued reconfigurations. This approach reduces the number of configurations that have to be processed (i.e., the DAG size). Thus, before starting the reconfig

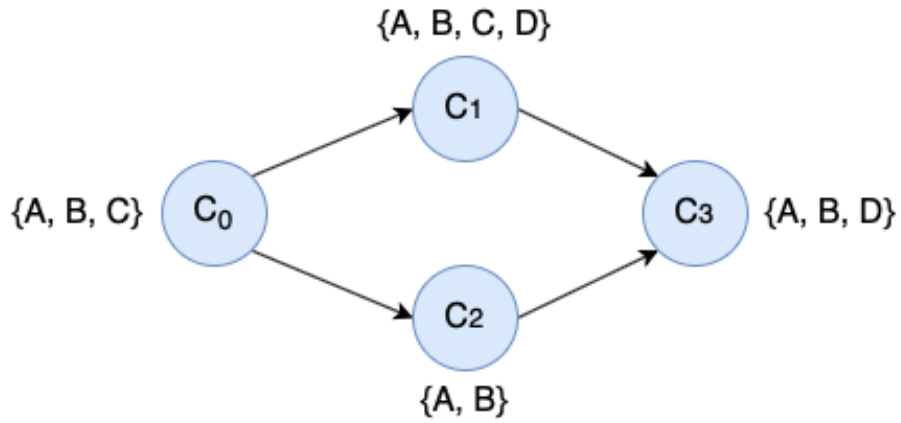


Figure 24: Example DAG of views.

algorithm, clients put their proposals in an external generalized lattice agreement (LA) service (GLA). GLA has an operation $la - propose(r)$, that takes all these proposals as argument and returns a configuration that is the merge (i.e., a lattice join) of all these competing proposals. Then, the clients take the configuration output of the lattice agreement and use it as their proposal in the reconfig.

4.1.2.4 Algorithm SpSn-Store

SpSn-Store [40] uses a Speculating Snapshot algorithm (SpSn) to reduce the worst case communication complexity of reconfigurations and operations concurrent with reconfigurations. In speculating snapshot, SpSn-Store can combine concurrently proposed changes and all established successors are ordered. Again, this algorithm is based on MWMR ABD.

Every configuration of this algorithm uses a SpSn. Clients may receive different configurations in response to their proposals when they invoke a SpSn, which prompts them to speculate on different configurations. The algorithm's goal is to keep track of the various configurations that clients suggest and combine them into a single configuration.

When a client invokes speculating snapshot, it goes through many rounds of message exchanges, each with two phases. A client disseminates its own changes in the first phase and collects changes proposed by others in the same round. In the second phase, if no other proposals detected, the client commits its proposal. A committed value represents an established configuration. If other changes are proposed, the client disseminates all of the changes gathered in the second phase. The client also collects values committed by other processes. Finally, the client starts a new round, proposing the combination of all changes observed in previous rounds.

4.1.2.5 Algorithm Ares

ARES [41] is reconfigurable, MWMR, crash-tolerant, erasure code based atomic algorithm. ARES is composed of three main components, (i) a reconfiguration protocol, (ii) a read/write protocol and (iii) a set of data primitives (DAPs).

A *configuration* in Ares is a data type that describes (*i*) the finite set of servers and how the servers are grouped into sets, i.e., *quorums*, s.t. each pair of quorums intersect, (*ii*) the consensus instance that is used as an external service to determine the next configuration, and (*iii*) the DAPs used.

Similar to traditional implementations, Ares uses $\langle tag, value \rangle$ pairs to order the operations on a shared object. In contrast to existing DSMs, Ares does not explicitly define the exact

methodology to access the object replicas. Rather, it relies on three, so called, *data access primitives* (DAPs): (i) the get-tag primitive which returns the tag of an object, (ii) the get-data primitive which returns a $\langle tag, value \rangle$ pair, and (iii) the put-data($\langle \tau, v \rangle$) primitive which accepts a $\langle tag, value \rangle$ as argument. To demonstrate the flexibility of DAPs, two different atomic shared R/W algorithms were expressed in terms of DAPs: (*i*) ABD-DAP: expressing the well celebrated ABD algorithm (cf. Section 4.1.1.1); (*ii*) EC-DAP: expressing an $[n, k]$-MDS erasure coding algorithm (e.g., Reed-Solomon [47]). This DAP splits the object into k equally sized fragments. Then erasure coding is applied to these fragments to obtain n coded elements, which consist of the k encoded data fragments and m encoded parity fragments. The n coded fragments are distributed among a set of n different servers. Any k of the n coded fragments can then be used to reconstruct the initial object value. As servers maintain a fragment instead of the whole object value, EC based approaches claim significant storage benefits.

The read, write, and reconfig operations in ARES are expressed in terms of the *DAP*. This provides the flexibility for ARES to use different implementation of *DAPs* in different configurations without changing the basic structure of ARES. A write (or read) operation π by a client p is executed by performing the following actions: (i) π invokes a read-config action to obtain the latest configuration sequence *cseq*, (ii) π invokes a get-tag (if a write) or get-data (if a read) in each configuration, starting from the last finalized to the last configuration in *cseq*, and discovers the maximum τ or $\langle \tau, v \rangle$ pair respectively, and (iii) repeatedly invokes put-data($\langle \tau', v' \rangle$), where $\langle \tau', v' \rangle = \langle \tau+1, v' \rangle$ if π is a write and $\langle \tau', v' \rangle = \langle \tau, v \rangle$ if π is a read in the last configuration in *cseq*, and read-config to discover any new configuration, until no additional configuration is observed.

4.1.3 Comparison

As we mentioned above ABD was the first crash-tolerant emulation of atomic shared memory, in an asynchronous message passing system. It has led to the common belief that "atomic reads must write". This belief was refuted by Dutta *et al.* [31] who showed that it is possible, under certain constraints, to complete reads in one round-trip. Several works followed, presenting different ways to achieve one-round reads (e.g., SLIQ, CWFR, OHSAM, OHMAM, FAST, SFW, SF, CCHYBRID, OHFAST, ERATO). SFW is the first algorithm that allows fast write operations in the MWMR setting. The ABD algorithm was extended by Lynch and Shvartsman with the MWMR variant of it. Then, the MWMR version of ABD was enhanced to prevent the versioning, building the algorithm COABD. Writers attempt to write a "versioned" value on the object. If the reported version is older than the latest, then the write does not take effect and it is converted into a read operation, preventing overwriting of a newer version of the object.

The above works on shared memory emulations were focused on small-size objects. On the other hand, the LDR algorithm can cope with large-size objects (e.g. files). However, the whole object is still transmitted in every message exchanged between the clients and the replica servers. Furthermore, if two writes update different parts of the object concurrently, only one of the two prevails. As we can see, there is no work in a static, crash-tolerant system that takes into account large objects, coverability, and handles concurrent updates on different parts of the file.

Early implementations of reconfigurable storage systems, such as RAMBO rely on consensus. When several reconfiguration requests are issued concurrently, the clients can use consensus to agree on the next configuration and then transfer the state of the object to this new configuration. However, it was later discovered that replicated service that do not require consensus can be reconfigured in a purely asynchronous way. DYNASTORE was the first asynchronous implementation of reconfigurable storage. Lately, a lot of progress was made in defining abstract

asynchronous reconfiguration in a simple and efficient way. Examples of works that improved in terms of efficiency, simplicity and modularity are the SM-Store and SpSn-Store. Ares shares similarities with consenus algorithms like RAMBO. Therefore, RAMBO uses consensus to choose only one successor for every configuration, in DynaStore configurations can have multiple successors, while *lattice agreement* or *SpSn* can ensure that configurations are ordered. i.e., for two configurations, the changes realized in one of them is also part of the other. Similar to DynaStore, Ares also requires reading of reconfiguration information (more than once in some cases) for each read and write operation. However, Ares is the only algorithm to combine a dynamic behavior with the use of erasure codes, while reducing the storage and communcation costs associated with the read or write operations. Moreover, in Ares the number of rounds per write and read is at least as good as in any of the above algorithms. Yet, the need to effectively handle large objects remains.

In [48], some of the above algorithms, including SM-Store, RAMBO, DynaStore and SpSn-Store, are implemented and evaluated both on the reconfiguration latency and the overhead these reconfigurations impose on concurrent operations. The results show that RAMBO and SM-Store have lower read latency than DynaStore and SpSn-Store, since these algorithms do not complete concurrent reconfigurations. However, for two or more concurrent reconfigurations, SM-Store has better read latency than RAMBO, since it batches multiple reconfigurations (and hence fewer configurations have to be traversed by reads). Regarding the reconfiguration latency, the latencies of RAMBO and DynaStore increase dramatically due to a lack of batching. In [41], there is an initial experimental evaluation of Ares, which measures the latency of each read, write, and reconfig operations, and examines the persistence of consistency even when the service is reconfigured between conigurations that add/remove servers and utilize different shared memory algorithms. The results prove the feasibility of the Ares framework with the erasure coding (EC) based approach, demonstrating its correctness and comparing its performance with a traditional approach. More specifically, the authors constructed two different atomic storage algorithms in Ares: (i) an EC based algorithm, and (ii) the ABD algorithm. As a general observation, the algorithm with the EC storage provides data redundancy with a lower communicational and storage cost compared to the ABD storage that uses a strict replication technique.

4.2 Byzantine-tolerant DSMs

An algorithm is considered very robust when it provides high consistency guarantees and performance in the presence of asynchrony and arbitrary (Byzantine) failures of servers [23]. A simple distributed storage, like algorithm ABD, can tolerate f crash failures out of $n = 2f + 1$ servers, but it is not resilience to Byzantine failures. This is in fact the case for all the algorithms present in Section 4.1.

According to [23, 49], algorithm ABD can be transformed to handle reader Byzantine failures using an *authenticated* model. This model assumes self-verifying data using digital signatures. More specifically, the writer signs the $< ts, value >$ pair with its private key. Thus, a reader using the public key can verify that the writer did indeed generate and sign the data. Also the faulty servers cannot generate a value with a higher timestamp than the latest writer used. So the value that the reader finds with the highest ts is correct. However, the use of digital signatures is difficult in large systems due to the fact that they require setup and key distribution. However, more recent solutions use an *unauthenticated* model without the use of self-verifying data (i.e., digital signatures). Below, we report secure algorithms that can cope with arbitrary (Byzantine) failures.

4.2.1 Algorithm SBQ-L

Small Byzantine Quromus with Listeners (SBQ-L) [50, 23] is the first MWMR atomic optimally resilient storage that can tolerate Byzantine server failures. If a storage implementation needs just n servers to handle f server failures, it is said to be optimally resilient. Tolerating f failures requires at least $n \geq 3f+1$ servers in the event of Byzantine failures. The best threshold that a Byzantine Fault Tolerance solution can be achieved, due to a known impossibility by Lamport [6]. SBQ-L uses more than $n \geq 2f + b + 1$ ($3f + 1$, assuming $b = f$) servers, where f is the number of crash failures and b is the number of Byzantine failures. SBQ-L copes with Byzantine failures using two main ideas. First, readers return the highest $< ts, value >$ when this is confirmed by $f + b + 1$ servers. This avoids the ABD's write-back phase but relies on servers to propagate data in case of client failures. Second, servers maintain a list of pending readers, called Listeners. If a read operation cannot obtain $n - f$ confirmations of the same value, servers add the reader to their Listeners list. Servers send all the concurrent updates to all the readers in the Listeners list.

The drawback of SBQ-L is that the timestamp might be arbitrary large and might exhaust the value space for timestamps. This issue is addressed using non-skipping timestamps, where writers choose $b+1^{st}$ highest timestamp. However, this solution sacrifices the optimal resilience of SBQ-L. It requires at least $2f + 2b + 1$ ($4f + 1$) servers.

4.2.2 Algorithm ACKM

The algorithm by Abraham, Chockler, Keidar, Malkhi, referred as ACKM [51, 23], is a SWMR algorithm that tolerates Byzantine client failures. It has two variants, a SWMR, wait-free, safe storage, and a SWMR, finite-write termination, regular storage. Both variants are optimally resilient. The write operation of ACKM has two rounds. In the first round the writer stores the $< ts, value >$ pair in the servers' pw fields. In the second round, it stores the same $< ts, value >$ pair in the w fields of the servers. In each round, the writer waits $n - f = 2f + 1$ servers' acknowledgments, where f is the number of crash failures. The read operation needs at least $f+1$ confirmations of the same value to prevent faulty values. If the reader does not obtain these confirmations, the algorithm must invoke another read round. In the new read round, if the reader can get the needed confirmations for any of the values, it can return this value. To ensure only safety semantic, a read operation can return in a constant number of rounds, at most $f + 1$. If after these iterations, the reader cannot get the needed confirmations, it can return a Byzantine value by safety. On the other hand, to ensure regular safety, the reader is guaranteed the needed confirmations once the writer stops writing, ensuring finite-write (FW) termination.

4.2.3 Algorithm GV

Unlike Algorithm ACKM, the Guerraoui and Vukolić optimally resilient storage algorithm, referred to as GV, enables reads to complete in only two rounds. This MWMR algorithm [52, 23] uses *high-resolution* timestamps ($HRts$) to include information about readers' logical time in the timestamps. In fact, $HRts$ is a two-dimensional matrix of timestamps with entries for each reader and server. The readers store their own read timestamps to a matrix in the servers. Writers use these timestamps to provide their timestamp with a much higher resolution. In the first write round, the writer stores its $< ts, value >$ pair, where the ts is its own low-resolution timestamp. Also, it gets copies of the readers' timestamps in servers's matrices, and create the high-resolution timestamp, $HRts$. In the second round, the writer writes the $< HRts, value >$ pair to servers.

4.2.4 Byzantine-Tolerant Reconfigurable DSM

Previous reconfigurable storage systems mentioned in Section 4.1.2 assume crash failures only. The work in [44] addresses the challenge of implementing an asynchronous reconfigurable storage tolerating Byzantine faults, where both replicas and clients may expose Byzantine (malicious) behaviour.

There are several difficulties that the proposed protocol addresses in order to implement asynchronously reconfigurable objects in the presence of Byzantine failures. The first problem is the *state transfer*, that is, to transfer the state of the object between configurations. The proposed solution is the following: during the reconfiguration procedure, each replica of the new configuration contacts a quorum of replicas in some prior configurations. These replicas then respond with their current state. It is worth mentioning that when a correct replica replies, it promises that it will no longer serve client requests in the old configuration. If in a later stage a client tries to perform an operation in the old configuration, the correct replicas that participated in the reconfiguration will refuse to interact with the client. Thus, the client will not able to complete the operation in the old configuration, but it will later discover the new configuration and will repeat its request.

An additional difficulty arises when the replicas of the old configuration turn Byzantine. They must refuse the operations in the old configuration even in that case! This case where a client tries to communicate with the old configuration is called *"still work here"* attack. This attack is solved by using forward-secure digital signatures [53]. More specifically, each configuration is assigned a number corresponding to the number of updates in the configuration; that is called the "epoch number". Each process has a sequence of private keys associated with epoch numbers. The forward-secure digital signatures' scheme allows only state transfer from a configuration with a smaller epoch number to a configuration with a larger epoch number. When a replica requests state from an older configuration the replicas of the old configuration update their private keys before replying. Thus, if the replicas of the old configuration turned Byzantine they cannot recover their private keys for that configuration and so cannot serve requests. This happens since the process can only sign messages with epoch numbers at least as large as the epoch number of the private key.

There is another attack, called *"slow reader"* attack. This attack is only possible when the reader's request is concurrent with a reconfiguration. For example, the reader tries to read the data and unfortunately it reaches replicas which are Byzantine or outdated, but at the same time it delays to send the request to the last correct replica of the quorum. Concurrently a configuration happens and all correct replicas update their private keys and cannot reply to the client requests in the old configuration anymore. However, a previous correct replica that the reader tries to send the message retain its private key for the old configuration since in the course it is corrupted by a Byzantine fault. Then, the reader reaches this corrupted replicas and get faulty data. This attack can be addressed by adding an extra round trip to confirm that the configuration is still relevant. More specifically, the reader sends a message to the replicas and asks them to confirm the digital signature that they still process.

Finally, in order to resolve conflicts between concurrent reconfiguration proposals in the face of asynchrony, they lead to a lightest agreement, mentioned as *configuration lattice*. A configuration of the lattice is a set of elements where each element has the form $(+, p)$ or $(-, p)$, where the first means that replica p has been added to the configuration and latter means that the replica has been removed.

4.2.5 Comparison

The above algorithms that can cope with Byzantine faults vary in consistency semantics, resilience (number and types of failures tolerated), and complexity (latency, for example). Algorithm SBQ-L can cope with Byzantine failures on servers, while on clients it can tolerate only crash failures. This algorithm assumes that the written value eventually is propagated to all correct servers. This happens either by the writer or by active propagation among the servers. However, in case where the writer fails and there are no active servers to propagate the written value, it is not certain that the value will eventually be written to correct servers. As a result, Algorithm ACKM is implemented to tolerate client failures, either by a SWMR, wait-free, safe storage or by a SWMR, FW-terminating, regular storage. Algorithm GV is constructed to optimize the performance of algorithm ACKM, by enabling reads to be completed in only two rounds.

Unlike the previous Byzantine-tolerant algorithms built for the static environment, the last reported algorithm built for the dynamic environment. So this reconfigurable algorithm has more difficulties when trying to cope with Byzantine faults. This algorithm uses digital signatures to deal with writers that try to write to an old configuration. Also it prevents the reader from learning about a malicious value from an old configuration, by invoking an extra round trip to confirm that the configuration is still relevant.

4.3 Discussion

In this chapter we examined several distributed shared memory (DSM) emulations that differ in the availability, consistency, complexity, the type of fault-tolerance, and their performance.

Algorithm ABD was a remarkable replication-based algorithm which implemented atomic (linearizable) read/write object in asynchronous environment using logical timestamps paired with values to order the operations. However, the replication-based strategy that ABD used incurs high storage and communication costs; an object copy is communicated to the majority of servers during each read/write operation. In addition, multiple communication rounds deemed necessary to ensure that each operation will obtain a consistent value of the distributed object. Multiple algorithms, which we have already described, extended ABD, trying to improve the communication overhead while imposing a small computation overhead. In some cases, restrictions on the underlying participation was also necessary for achieving specific algorithmic performance. The problem of keeping copies consistent becomes even more challenging when failed servers need to be replaced or new servers are added, without interrupting the service. For this purpose we dealt with reconfigurable algorithms.

However all the above solutions assume crash failures only. Thus we also considered the Byzantine fault model, where replicas and clients may expose arbitrary or even malicious behaviour. We concluded describing one aynchronous reconfigurable storage tolerating Byzantine faults.

The consistency and availability of the replicated data in a dynamic environment has been thoroughly researched for the crash fault model. However, there is little literature on reconfiguration in the presence of Byzantine failures. This happens since there are more challenges in a Byzantine fault-tolerant reconfiguration system than in a crash fault-tolerant, which does not allow simply the simple adaption of dynamic techniques proposed for the crash failure model to the dynamic model with the presence of Byzantine faults.

The advantage of these memory emulations is that they come with *provable guarantees*. However, all the above algorithms required to transmit the entire object over the network per read and write operation. Moreover, if two concurrent write operations affected different "parts" of the object, only one of them would prevail, despite the updates not being directly conflicting.

In such settings, large-sized objects are difficult to handle. Even more challenging is to provide linearizable consistency guarantees to such objects. There are dozens of distributed storage systems that exist on the market today that can handle large objects. However, there is no (known) trace of Byzantine fault tolerant inside them, since the cost of handling this type of fault is excessive for a commercial system. Such systems is the subject of the next chapter.

5 Distributed Storage Systems

As discussed, although the above distributed storage emulations managed to meet the requirements of strong consistency, fault-tolerance, availability, and reliability of the data, they are not able to respond to the new requirements in terms of volume of data and high performance. In this section, we discuss a set of the main characteristics used in the market to build distributed storage systems in order to meet the big data requirements. These characteristics include scalability, data availability, reliability and security. Dozens of storage systems exist in the market today. Below we choose to focus on the most prominent ones and discuss their strengths and limitations. Some of them have been proposed to meet the growing demand for scalable and efficient data management, some of them coming from the data-intensive distributed computing sector, and others have been explicitly intended for Cloud.

Table 3 presents a comparison of the main characteristics of storage systems that we will discuss in this chapter, and that can be summarized as follows:

- *Data Scalability*: the system can manage the growing data requirements, by expanding the current infrastructure with more resources or disks.
- *Data Access Concurrency*: how the system manages conflicting accesses to data.
- *Consistency guarantees*: the consistency model that the system follows.
- *Versioning*: whether the system records versions of changed data.
- *Data Striping*: whether the system divide the data across the set of disks.
- *Reconfiguration*: whether the system change the configuration of the system as it executes.

5.1 GFS

The Google File System (GFS) [54, 10] is a scalable distributed file system developed by Google to meet Google's data storage and usage needs. The last version of Google File System codenamed Colossus was released in 2010. A GFS cluster consists of a single Master server and multiple Chunkservers and can be accessed by multiple clients, as shown in Fig. 25. If any of the three nodes (master, primary replica, or secondary replica) fails, the system can be unavailable for a short time until it is replaced [62]. Thus, GFS guarantees non-blocking reconfiguration but it experiences short downtime.

The files in GFS are divided into chunks of fixed size. The master server assigns a unique 64-bit chunk handle (i.e., identifier) to each chunk when is created. All chunks are stored on local disks at chunkservers which are responsible both for read and write data on them. For reliability, each chunk is replicated on multiple chunkservers ($default = 3$).

The master maintains the file system metadata of the GFS cluster. This includes the file and chunk namespaces, the mapping from files to chunks, and the locations of each chunk's replicas. All these metadata are kept in the master's memory, making all operations faster.

Table 3: Comparative table of storage systems.

System	Data scalability	Data access Concurr.	Consist. guarantees	Versioning	Data Striping	Non-blocking Reconfig.
GFS [54]	YES	concurrent appends	relaxed	YES	YES	YES (short downtime)
Colossus [55]	YES	concurrent appends	relaxed	YES	YES	YES
HDFS [56]	YES	files restrict one writer at a time	atomic centralized	NO	YES	YES
Cassandra [57]	YES	YES	tunable (default= eventual)	YES	NO	NO
Dropbox [58]	YES	creates conflicting copies	eventual	YES	YES	N/A
Redis [59]	YES	YES	eventual	YES	NO	NO
Blobseer [60]	YES	YES	atomic centralized	YES	YES	YES
Tectonic [61]	YES	files restrict one writer at a time	Read-your-writes	YES	YES	YES

The master also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers. The master periodically communicates with each chunkserver in HeartBeat messages to give it instructions and collect its state.

GFS offers a relaxed consistency model that supports Google highly distributed applications, but is still relatively simple to implement. Most files are mutated by appending new data rather than overwriting existing data. Also GFS supports record append at the offset of its own choosing rather that written at an application-specified file offset as in the write case. This operation allows multiple clients to append data to the same file concurrently without extra locking. The namespace mutation operations on GFS are atomic and are treated exclusively by the master . The namespace locks guarantee its atomicity and accuracy. The status of a file region after a data transfer depends on the type of mutation, the success or failure of the mutation, and the existence or not of concurrent mutations. As shown in Fig. 26, a file region is consistent if all clients see the same data, independent of the replica they access. A file region is defined if it is consistent and all clients see the modification in its entirety, as if change were atomic. Concurrent accesses leave the region consistent, but possibly undefined. Failed writes leave the region inconsistent. GFS uses chunks version numbers to detect any replica that has become stale because it missed mutations. Stale replicas will never be used and will be destroyed by a garbage collector at the earliest opportunity.

In order to handle concurrency, it has only one master node for the file system metadata which may lead to a high congestion or a bottleneck resulting from a massive concurrent requests.

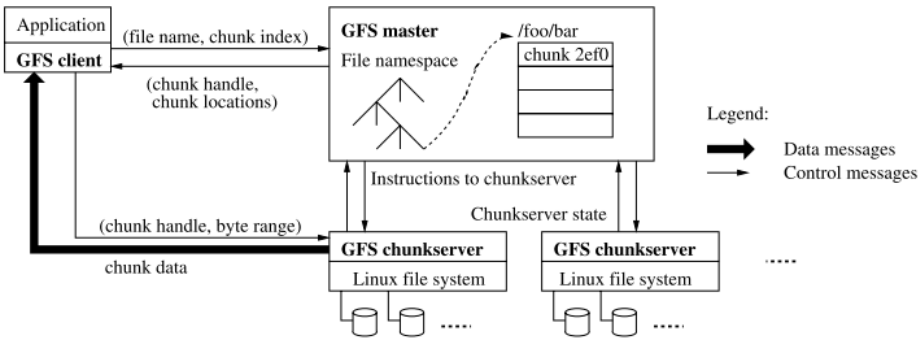


Figure 25: GFS Architecture [54]

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Figure 26: File Region State After Mutation [54].

However, Google tried to minimize the impact of this weak point by replicating the master on multiple backup copies called "shadows". This also increases the fault-tolerance of the system.

5.2 Colossus

Colossus [55] is the latest version of the GFS, released in 2010. Like its predecessor, Colossus is optimized for working with large data sets, scales perfectly, is a highly accessible and fault-tolerant system, and also allows you to reliably store data. It supports Google products like YouTube, Drive, and Gmail. Colossus is the one of the three building blocks used for implementing Google's storage services, such as Firestore, Cloud SQL, and Cloud Storage. The other two are Spanner and Borg. Spanner is Google's globally consistent, scalable relational database, and Borg is a scalable job scheduler that launches everything from compute to storage services. Borg was and continues to be a big influence on the design and development of Kubernetes. Thus, each storage service uses these three building blocks to provide everything you need. Borg provisions the needed resources, Spanner stores all the metadata about access permissions and data location, and then Colossus manages, stores, and provides access to all your data.

Colossus was designed to solve the GFS problems, including the slow responses caused by the batch processing, the unsuitable 64MB-chunk size, and the potentially bottleneck resulting from a massive concurrent requests. The architecture of Colossus, as shown in Fig. 27, contains five components, including the Client Library, Curators, Metadata database, D File Servers and Custodians. The Client library is used by an application or service to interact with Colossus. It implements features, such as software RAID and allows applications built on top of Colossus to use a variety of encodings to fine-tune performance and cost trade-offs for different workloads. Curators constitute the largest part of Colossus Control Plane. Clients send control operations to these curators, such as file creation. Curators store metadata in Google's high-performance NoSQL database, BigTable. Storing file metadata in BigTable allows Colossus to scale up

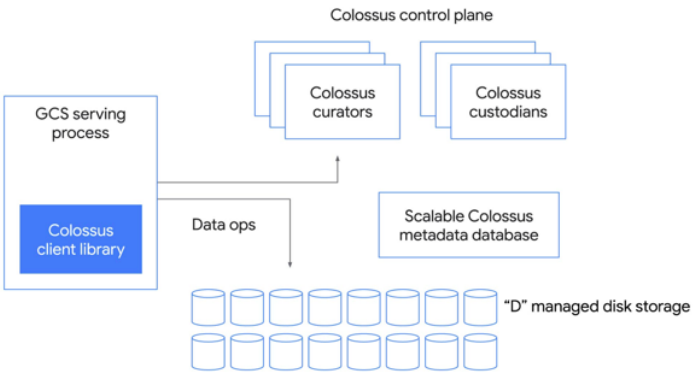


Figure 27: Colossus architecture [55]

by over 100x over the largest previous-generation clusters. The data flows directly between the client and the D file servers, which are network-attached disks. This minimizes the number of hops for data on the network. Custodians are background storage managers which operate on the file servers, maintaining the durability, availability of data and the efficiency. Also, Colossus can be scaled dynamically by adding or removing cluster nodes without restarting, so it provides non-blocking reconfiguration.

5.3 HDFS

The Hadoop Distributed File System (HDFS) [56] is a distributed file system component of the Hadoop ecosystem. HDFS is now a subproject of Apache Hadoop, a very popular Big Data system that allows distributing the processing of large data sets across clusters of computers. HDFS is highly fault-tolerant and it is designed to run on low-cost hardware. As shown in Fig. 28, it stores metadata, including the locations of data blocks (the mapping of blocks to DataNodes), on a server called NameNode and application data on servers called DataNodes. It stores each file as a sequence of blocks (data striping) and replicates each block for fault-tolerance. The number of DataNodes can be dynamically increased, providing non-blocking reconfiguration.

When a file has been closed successfully, it becomes immediately visible to other clients, that is, HDFS supports linearizability (atomic) consistency. However, the cache in HDFS is centralized since it is managed by one NameNode. The HDFS cluster's NameNode is the primary server that manages the file system namespace and controls client access to files. It provides concurrency by restricting the file access to one writer at a time.

Similarly with GFS, HDFS has one NameNode per cluster which periodically receives HeartBeat messages and a Blockreport (a list of blocks) from each of the DataNodes. To reduce the bottleneck effect that may happen resulting from a massive concurrent transaction stream in the unique NameNode in the cluster, NameNode batches multiple transactions initiated by different clients. The HDFS is designed more for batch processing rather than for online transactions. For the batch processing Apache Hadoop uses the Hadoop MapReduce framework.

5.4 Cassandra

Cassandra [57] is a NoSQL distributed database offering continuous availability, high performance, horizontal scalability, and a flexible approach with tunable parameters. It was initially developed by Facebook for Facebook's inbox search feature. Today, it is an open-source application of Apache Hadoop.

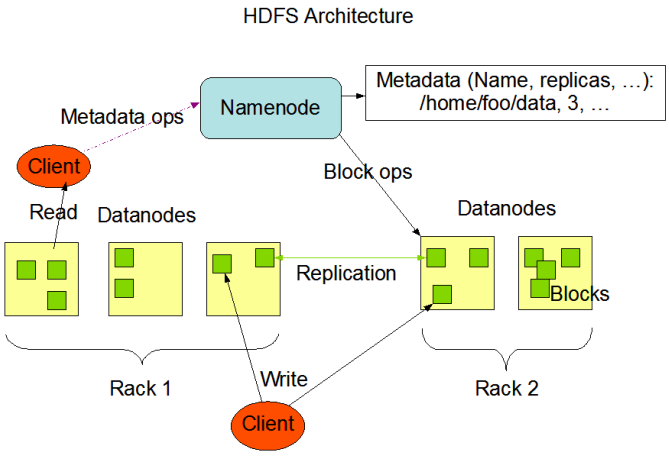


Figure 28: HDFS Architecture [56]

Cassandra uses peer-to-peer communication where each node is connected to all other nodes, forming a ring topology. The protocol used to achieve this communication is gossip, in which nodes periodically exchange state information about themselves. All the nodes in a cluster are equal and can serve read and write requests.

Cassandra splits the data into multiple partitions and then stores each partition in a node of the cluster. A data partition consists of all the rows that have the same partition key. A partition key is used to determine which node to store data on and where to find data. When data arrives in the cluster, the partitioner applies a hash function to the partition key and coverts the partition key into a token. Each node in a cluster owns a range of tokens. Thus, the token determines which node will get the data partition.

For example, in Fig. 29, the output of the partition function when the partition key is equal to "CA" is 12. Next, the coordinator, which can be any node in the cluster, searches for which node has the token 12, and it forwards all the data partition to that node. In the current example, it sends to that node the second and third rows.

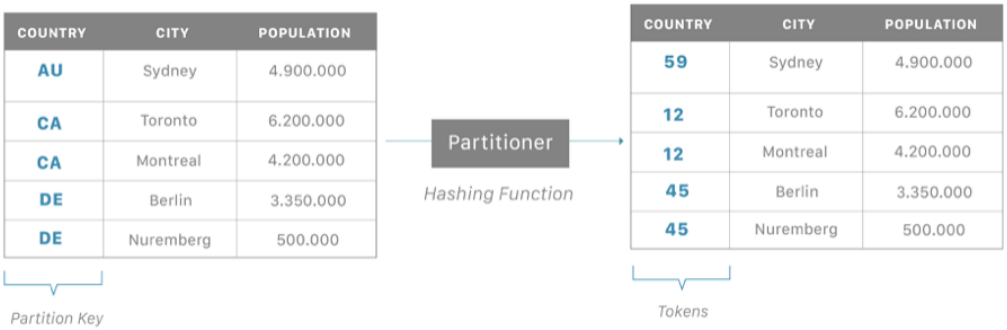


Figure 29: Cassandra partitioner [57]

Cassandra is highly scalable, and you can increase performance just by adding more nodes. By default, it guarantees eventual consistency, which implies that all updates reach all replicas eventually. This means that Cassandra chooses availability over consistency in order to achieve lower latency. This is due to the fact that it is used in web based applications which server large number of clients and data. However, Cassandra offers tunable consistency for read

and write operations, so that the system can guarantee weaker or stronger consistency, as required by the client application. By weaker consistency we mean that the system would be highly available and partition-tolerant. While, by stronger consistency we mean that the system would be consistent and partition-tolerant. The required consistency can be achieved by tuning the consistency level (CL) and the replication factor (RF) parameters. RF specifies how many copies of a store object (i.e., a row in Cassandra's Database) is kept among the participants. Given the value of the RF, the CL controls how many responses the coordinator waits for before the operation is considered complete.

Let's look at some of the CL examples for the write operation to be successful. For example, $CL = ONE$ means it needs acknowledgment from only one replica node. Since only one replica needs to acknowledge, the write operation is fastest in this case. However, $CL = QUORUM$ means it needs acknowledgment from a majority of replica nodes across all datacenters.

Now let's see some RF examples. In the previous example (Fig. 29), the replication factor is one ($RF = 1$) which means that there is only one copy of each row in the Cassandra cluster. However, many copies of each data partition can exist in the database. For example, if $RF = 3$, the data must be stored on three replica nodes. This means that there are three nodes that covers each token range. With this replication technique, Cassandra ensures reliability and fault-tolerance. Also, it can distribute the jobs among the nodes, resulting in better performance.

To understand the Cassandra CL factor versus RF consider the following. When $R + W > RF$, where R is the read consistency level, and W is the write consistency level, strong consistency can be guaranteed. This guarantees that at least one of the replicas will participate in both the write and read requests, and so the write will be visible to the reader. For example, if $RF = 3$, then the addition of read and write consistency levels must be at least 4. A case would be the coordinator having to wait for 2/3 replica responses for both operations. However, if the user needs a faster read or write operation, the CL of that operation must be decreased and the other one increased so that the strong consistency remains valid. On the other hand, if the above condition is not true, so that $R + W \leq RF$, then the system can provide eventual consistency. To conclude, in order to guarantee atomicity, if n is the total number of available replicas and RF is n, then $n/2 + 1$ must respond.

Finally, Cassandra allows the removal and addition of a single node at a time, in contrast to the reconfigurable algorithms mentioned in the previous chapter, that allow a complete modification of the configuration in a single operation.

5.5 Dropbox

Dropbox [58, 63] is one of the most popular commercial cloud-based file synchronization service on the market, including Microsoft OneDrive, Google Drive and other, with big appeal to end-users and continues to update its sharing features. It provides eventual consistency, synchronising a working object for one user at a time. Thus, in order to access the object from another machine, a user must have the up-to-date copy; this eliminates the complexity of synchronization and multiple versions of objects.

File synchronization services propagate local changes asynchronously, and often provide a means to restore previous versions of files. Furthermore, they are only loosely integrated with the file system, allowing them to be portable across a wide range of devices. A cloud-based file synchronization service has two parts: server software that stores user data and a client-side daemon that actively synchronizes one or more directories with the server. Dropbox is an example of such a server software. User metadata is stored in the company's data centers, while the actual files are stored on Amazon's S3 cloud service.

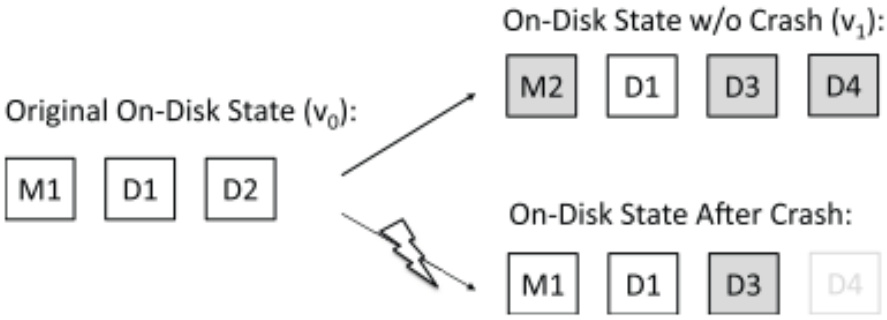


Figure 30: An example of Dropbox's inconsistency [58]

In order to detect remote changes, Dropbox uses push-based notifications. On the other hand, the detection of local changes is based on monitoring services, such as Linux's *inotify*. To detect offline changes, Dropbox keeps detailed metadata for each file, including last modification and attribute change times. When the system goes online, Dropbox scans its monitored files and uploads the new version of a file if its metadata has changed (even if the new modification time precedes the recorder one). However, it keeps revisions of all files for up to thirty days, allowing the user to revert to a previous version, in case the upload was a result of a data corruption or any other possible similar reason.

Dropbox uses a deduplication scheme to eliminate the reduntant data. It breaks files into 4MB-chunks, creates a hash of each chunk and compares those hashes to what they have stored in the cloud. If there are any hashes that don't present to the destination, Dropbox uploads that chunk. If the hash matches one already stored online, this chunk is not sent. For partially modified chunks, Dropbox uses rsync to transmit only the changed portions. Finally, to achieve integrity at the client, Dropbox downloads the chunks of files to a staging area, assembles them, and then renames them to their final location.

Despite Dropbox's high capabilities, it faces serious issues with data consistency. One main issue that Dropbox faces is that because it uploads asynchronously, generally at a much slower rate than the local disk, there is no guarantee that the current version of the file stored in Dropbox reflect the latest local version. For recovery, Dropbox uses a journaling mechanism. It ensures that the data blocks are written first and then the journal metadata (write them in a log before commit them to a fixed location, allowing for replay or rollback). Fig. 30 depicts an example where inconsistency may be raised in Dropbox. The original state of the file on disk is represented as v0 with an inode M1 pointing to two data blocks, D1 and D2. Then, a new application overwrites the last block D2 with D3 and appends a new data block D4 resulting to a new inode M2. If there is no crash, the final state of the file will be v1 containing all data blocks. If however a crash occurs before M2 is journaled, but after D3 and D4 are written, the file will be inconsistent as shown in the lower right of the figure. Thus, the metadata of the file is consistent on disk at v0 but the file's data is an inconsistent mixture of v0 and v1.

5.6 Redis

Redis [59] is an open source, in-memory key-value store. The read/write response time for Redis is extremely fast since all the data is in memory. As shown in Fig. 31, Redis is based on a Master-Slave architecture, i.e., it enables replication of master Redis instances in replica Redis instances. Since the replication process is asynchronous, each replica's copy could be out of date. This makes Redis to ensure eventual consistency. The number of read replicas can

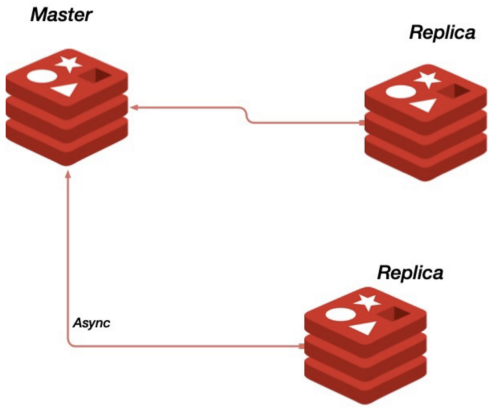


Figure 31: Redis architecture [64]

be dynamically increased or decreased. However, if the master goes down, the writes will be blocked because the replica nodes are read only. Either you must add a new node, losing all the old data, or make any existing replica the new master. Thus, it provides reconfiguration but not in a non-blocking manner.

The use of Redis is rather easy; Redis will internally store the key and value when users execute commands like set key value. Redis returns the value with a simple get key command from the user. It supports a variety of data structures, including strings, lists, sets, sorted sets, hashes, bitmaps, and HyperLogLogs, avoiding the conversion from one data type to another when the data is loaded into Redis. The data size cannot exceed the main memory limit because all the data are in main memory. Redis starts to reply with an error to write commands when the max memory limit is reached.

Redis has two persistence mechanisms: RDB (Redis Database) and AOF (Append Only File). RDB persistence provides point-in-time snapshots of the database at specified intervals. AOF persistence logs every write operation. When the database server starts, Redis reads the AOF log to reconstruct the database. RDB is perfect for backup, but if RDB stops working all data changes since the last snapshot are lost. In comparison, AOF has better durability, although adopting AOF persistence may result in performance loss.

Redis has a command called "WAIT" in order to implement synchronous replication. This command blocks the current client until all the previous write commands are successfully transferred and acknowledged by at least the specified number of replicas. Redis provides eventual consistency. Even though a write may wait until all replicas reply, reads do not wait and always terminate as soon as they receive messages from the master.

5.7 Blobseer

Blobseer [60] is a project of KerData team, INRIA Rennes, Brittany, France. It is a large-scale distributed storage system developed to handle heavy access concurrency. The data stored in Blobseer is a long sequence of bytes called BLOB (Binary Large Object). The architecture of Blobseer is illustrated in Fig. 32. It includes Metadata and Data providers. It has a single Provider manager which keeps information about the available storage space and assigns available data providers to write requests. A Version manager, the key component of the system, deals with the serialization of the concurrent write requests and assigns a new version number for each write operation. Unlike GFS and HDFS, Blobseer does not centralize metadata on a single machine, but it uses a centralized provider manager and version manager. Thus, linearizability is easy to achieve. Also, Blobseer implements a dynamic configuration

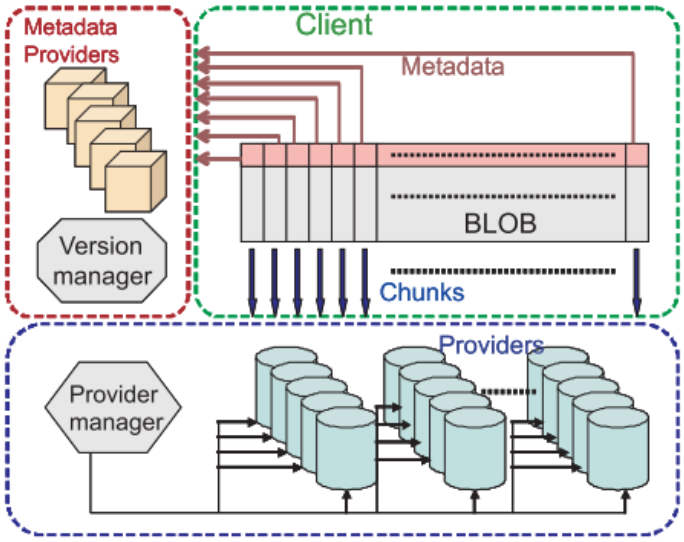


Figure 32: BLOBSEER Architecture [60]

framework to automatically adjust the number of storage nodes.

This system uses data striping and versioning, which allow writers to continue editing in a new version without blocking other clients that use the current version. In a write operation, each BLOB is stored as chucks over the storage nodes, using a data striping technique. After all chunks have been successfully sent, a versioning mechanism via the version manager is used in order to assign a consistent version to the Blob and adds the WRITE into a queue containing in-progress WRITES. In a read operation, the reader requests the metadata of a specific BLOB id and BLOB version, and then it retrieves the chunks in parallel from the storage nodes.

Regarding their fault-tolerant model, they assume permanent or amnesia crashes [65], which are both types of crash failures. A permanent crash is characterized by a lack of response that begins at a certain time and lasts an endless period of time. An amnesia crash is a lack of reaction that begins at a specified time and lasts for a short period of time until the entity returns to the stable state it was in prior to the crash.

5.8 TECTONIC

TECTONIC [61] is the current consolidated distributed file system of Facebook that manages the data for an entire data center, scaling to Exabytes in size. TECTONIC guarantees read-after-write consistency, also known as read-your-writes, for data and directory operations. Before TECTONIC, Facebook used a constellation specialized storage systems to support different workloads and applications. Two large-scale use cases (storage tenants) at facebook are the blob storage and data warehouse storage. The Blob storage is responsible for storing and serving photos and videos shared across facebook. While the data warehouse supports data processing and analytics which are important to build and deliver user's products like Search, Newsfeed and Ads.

Before TECTONIC, blob storage consisted of two specialized systems, Haystack and f4. More specifically, a blob was split across Haystack and f4. Haystack stored hot blobs with higher request rate but it was not storage efficient. On the other hand, f4 stored warm blobs with lower request rate but it did not support blob uploads. Data warehouse used HDFS because it is throughput efficient, but it is not suitable for small IO. However, a single instance of HDFS can

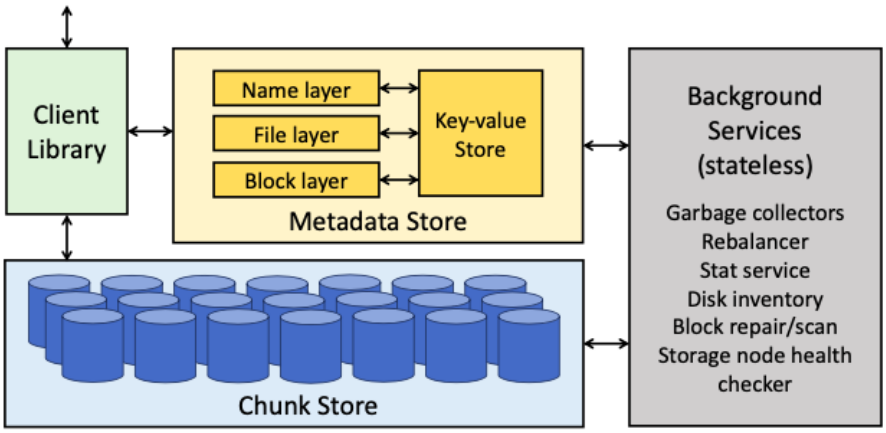


Figure 33: TECTONIC architecture [61].

only store tens of petabytes of data due to its single name node based design. As a result each data center had to use multiple HDFS instances to serve data warehouse. Similar to HDFS, TECTONIC can add and remove nodes dynamically, providing a non-blocking reconfiguration.

TECTONIC unifies Facebook's previous storage services (e.g. federated HDFS, Haystack, and f4). A single TECTONIC cluster can manage the data for an entire data center, scaling to Exabytes in size. As shown in Fig. 33, the architecture of TECTONIC is composed of four main components, the Chunk store, Metadata Store, Client Library, and Background Services, which allow us to achieve exabyte-scale clusters. At the core of TECTONIC is chunk store, a set of independent storage nodes which store and access data chunks on hard drives. TECTONIC files are divided into blocks. To provide durability blocks are further split into chunks using the Reed-Solomon encoded (RS encoded) [47]. Chunks are stored as file on the hard drives of the storage nodes. The capacity of the chunk store scales linearly with the number of storage nodes. On top of chunk store, there is a metadata store which is built on a linearizable scalable key-value store. Unlike the HDFS which stores the metadata on a single node, TECTONIC divides metadata into multiple layers and hash-partitions each layer (Name Layer, File Layer, and Block Layer). The Client Library orchestrates the file system by interacting with the different Metadata Layers and the Chunk Store. This independence between the components helps them scale independently. Client Library uses single-writer semantics, that is, it allows a single writer per file. Single-writer semantics avoids the complexity of serializing writers to a file from multiple writers. However, in the case a tenant needs multiple-writer semantics, it can build serialization semantics on top of TECTONIC. Finally, each cluster runs background services to maintain cluster consistency and fault-tolerance.

Let's see how the Client Library handles a read operation. First the Client Library gets the file id of filepath from the Name Layer. Then it gets the block list with the block ids of the file from File Layer. Then it fetches the location of chunks (i.e. list of disks) of those blocks from the Block Layer. Now it can read the data from storage nodes (Chunk Store). Once this stage is complete, it returns the data. Similarly, to write a block to a file, the Client Library gets the location of chunks from the Block Layer. Once it stores the chunks of the block to those storage nodes, it then saves the final chunk locations to the Block Layer and it adds the block to the block list in the File Layer. Finally, it returns success to caller.

5.9 Comparison

Scalability is the main concern that these systems manage to address. Those systems are designed to manage increasing amounts of data in an appropriate manner. The above systems are built to tolerate withstand failures, however they are not capable of dealing with the majority of Byzantine attacks. As discussed, Byzantine faults provide a number of difficulties for system designers and can restrict performance and require more hardware. Sever research works [66, 67, 68, 69], however, are attempting to harden these systems against byzantine faults in a more efficient, higher performance, and more scalable manner than previously thought possible.

The first-generation distributed file systems of Google and Facebook, that is, GFS and HDFS respectively, were not scalable enough because their centralized components do not allow them to support huge amount of metadata. Thus, both Google and Facebook introduced next-generation storage systems that provide a more scalable and highly available metadata system. Google continues with Colossus and Facebook with Tectonic. Both systems store their metadata in a key-value store. Google used its BigTable key-value store to store Colossus' metadata. Facebook chose the ZippyDB key-value store for Tectonic.

Other major concerns of DSSs are data availability and fault-tolerance. Some mechanisms to increase the availability of data and ensure fault-tolerance are replication mechanisms, versioning, snapshots, etc. In addition all the above DSSs (except Cassandra and Redis) use data striping to manage and optimize concurrent access to the data. Furthermore, Blobseer continues editing in a new version without blocking other clients who continue to use the current version transparently. Thus, saving data as BLOB combined with a data striping mechanism can greatly improve the performance of such a system and allow it to handle larger files.

Cassandra and HDFS are both projects of Apache. Unlike HDFS, Cassandra does not have a master-slave architecture; it follows a masterless architecture. HDFS, like the other systems mentioned, can handle large files, using the data striping technique. On the other hand, Cassandra can work with multiple small records (structured data, record-level indexing). If one wants to store large unstructured files in Cassandra, they first need to split them into multiple parts. Cassandra and Redis are two real-time databases. However, Redis has a significantly faster response time than Cassandra, since it saves a lot of data in-memory rather that storing it to disk, as Cassandra. On the other hand, Cassandra can guarantee considerably better fault-tolerance than Redis due to its architecture.

Regarding consistency guarantees, HDFS and Blobseer provide linearizability consistency. HDFS achieves linearizability by using centralized machine for metadata. Unlike HDFS, Blobseer does not centralize metadata on a single machine, but it uses a centralized version manager. Thus, linearizability is easy to achieve. On the other hand, Tectonic provides a type of causal consistency, more specifically read-your-writes consistency. This means that when a user performs changes, it can view theses changes (read data) right after making those changes. However, like its ancestor, HDFS, it allows a single writer per file. This simplifies the complexity of serializing writes to a file from multiple writers. Nevertheless, if a tenant needs multi-writer semantics, it will have to build its own write serialization semantics on top of Tectonic.

Dropbox is not even far relative of the other file systems. It just synchronize the files between local storage devices and the cloud for reasons like backup, share and access one's files from anywhere. On the other hand, to use the other systems as cloud storages, it is needed to build user-friendly and efficient systems as Dropbox. But they will neither be so user-friendly or efficient as Dropbox which guarantees a weaker consistency. There are limited number of papers [70, 71, 72, 73, 60, 61] that perform proof of concept experimental analysis of the above DSSs, in order to verify that the DSS will function as envisioned and not violate its guarantees.

6 Discussion

In the previous two chapters we examined the most popular consistency models and discussed about several distributed algorithms and storage systems implementing them, trying to provide high data availability and fault-tolerance. However, these DSSs either sacrifice performance or availability depending on the consistency model they rely on.

As discussed, systems implemented using strong or sequential consistency suffer in terms of performance. On the other hand, those systems using relaxed or eventual consistency, have serious issues when conflicting writes appear. In addition, most of these systems, like GFS for example, use centralized components, and this may lead to high congestion or to a bottleneck effect.

To our opinion the most appropriate model able to provide high consistency, concurrency and availability seems to be the Atomic / Linearizability Consistency model. However, despite the fact that there is a vast amount of theoretical work on this consistency model for more than 30 years [74], to the best of our knowledge there is still no system fully implementing it. Some previous attempts, like LDR [30] was promising, but they seem to suffer from communication delays and communication overheads since the whole object is still transmitted in every message exchanged between the clients and the replica servers. Furthermore, if two writes update different parts of the object concurrently, only one of the two prevails. In addition, those solutions cannot be found readily and were not adopted by commercial distributed storage applications. The most commercial DSSs we considered above, avoid providing strong consistency guarantees (such as atomicity) as they are considered costly and difficult to implement in an asynchronous, fail prone, message passing environment. The minority may use atomicity, but they succeed it using centralized components or restricting one writer at a time. Hence, such solutions either choose to offer weaker or tunable guarantees to achieve better performance when atomicity is not preserved, or they devise strategies to address the issue of consistency.

To the best of our knowledge, currently there is no distributed storage system that can provide provable atomic consistency guarantees in a decentralized environment in the absence of a failure detection and coordination; and this is our motivation: to create a DSS that will be proved to guarantee these characteristics. Our recent work [75] proposes a framework which uses a fragmentation strategy, aiming to reduce the communication cost of write/read operations by splitting data into smaller atomic data objects (blocks), while enabling concurrent access to these blocks. Based on that, we introduced new consistency guarantees, that characterize the consistency of the whole object, which is composed by smaller atomic objects (blocks). We are currently working on the implementation of an erasure-coding (EC) storage (such as Ares [41]) which divides the object into encoded fragments and deliver each fragment to one server. The object can be recovered from a subset of the fragments. However, operations are still applied on the entire object. Thus, we can evaluate the performance of the system by combining our fragmentation approach with EC. Ares also implements a Reconfiguration service in order to mask host failures or switching between storage algorithms without service interruptions. However, Ares may need to be extended to address these objectives. A question that must be answered is how such a DSS may compare to commercially used solutions? That is, no evidence exists to date to examine what the gains are from commercial solutions that adopt less than intuitive guarantees. In our next work, we will put DSS and chosen open-source, commercial solutions in a head-to-head comparison in order to answer the question: Is it worth to trade consistency for performance?

We also aim to use a failure prediction service to estimate the risk of a device failing in order to trigger the reconfiguration service. It could be based on a monitoring service that would collect S.M.A.R.T. (Self-Monitoring, Analysis and Reporting Technology) metrics of

the servers, indicating a possible drive failure. Machine Learning would be used to identify correlations on the metrics so to predict drive failures. We will integrate the failure prediction service with reconfiguration. Also, we need to specify the aggressiveness of reconfiguration: reconfiguring in every failure notification may result in many frequent reconfigurations, whereas waiting too long may disable the service due to many failures.

References

[1] "EMC digital universe." `https://www.cycloneinteractive.com/our-work/emc-digital-universe/`, last accessed on 08/10/2022.

[2] "IDC." `https://www.idc.com/`, last accessed on 08/10/2022.

[3] M. v. Steen and A. S. Tanenbaum, *Distributed Systems: Principles and Paradigms, 3rd ed.* 2017.

[4] P. Viotti and M. Vukolic, "Consistency in Non-Transactional Distributed Storage Systems," *ACM Computing Surveys (CSUR)*, vol. 49, pp. 1 – 34, 2016.

[5] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, no. 2, p. 374–382, 1985.

[6] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 382–401, 1982.

[7] V. Gramoli, N. Nicolaou, and A. A. Schwarzmann, *Consistent Distributed Storage*, vol. 20. 2021.

[8] "File storage, block storage, or object storage?." `https://www.ibm.com/cloud/blog/object-vs-file-vs-block-storage`, last accessed on 08/10/2022.

[9] "DFS." `https://cseweb.ucsd.edu/classes/sp16/cse291-e/applications/ln/lecture13.html`, last accessed on 08/10/2022.

[10] A. Elomari, L. Hassouni, and A. Maizate, "The main characteristics of five distributed file systems required for big data: A comparative study," *Advances in Science, Technology and Engineering Systems*, vol. 2, no. 4, pp. 78–91, 2017.

[11] P. K. Sinha, *Distributed Operating Systems: Concepts and Design*. Wiley-IEEE Press, 1st ed., 1996.

[12] E. Brewer, "Cap twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[13] "Data deduplication." `https://www.netapp.com/data-management/what-is-data-deduplication/`, last accessed on 08/10/2022.

[14] J. Stender, "Snapshots in Large-Scale Distributed File Systems," no. September, 2013.

[15] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[16] A. Fernández Anta, C. Georgiou, and N. Nicolaou, "CoVerability: Consistent Versioning in Asynchronous, Fail-prone, Message-passing Environment.," *In Proc. of NCA*, pp. 224–231, 2016.

[17] H. Attiya and J. L. Welch, "Sequential Consistency versus Linearizability," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 2, pp. 91–122, 1994.

[18] S. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[19] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, (Vancouver, BC), USENIX Association, Oct. 2010.

[20] D. B. Terry, "Replicated data consistency explained through baseball," *Commun. ACM*, vol. 56, pp. 82–89, 2013.

[21] L. Kuper and P. Alvaro, "Toward domain-specific solvers for distributed consistency," *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 136, no. 10, pp. 1–10, 2019.

[22] W. Vogels, "Eventually consistent," *Queue*, vol. 6, no. 6, pp. 14–19, 2008.

[23] G. Chockler, R. Guerraoui, I. Keidar, and M. Vukolić, "Reliable distributed storage," *IEEE Computer*, vol. 42, no. 4, pp. 60–67, 2009.

[24] L. Lamport, "On Interprocess Communication, Part I: Basic Formalism," *Distributed Computing*, vol. 1, no. 2, pp. 77–85, 1986.

[25] L. Lamport, "On interprocess communication - Part II: Algorithms," *Distributed Computing*, vol. 1, no. 2, pp. 86–101, 1986.

[26] M. Raynal, "Concurrent Programming: Algorithms, Principles, and Foundations," in *Springer Berlin Heidelberg*, 2013.

[27] N. Lynch and A. A. Shvartsman, "RAMBO: A reconfigurable atomic memory service for dynamic networks," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2508, no. June, pp. 173–190, 2002.

[28] L. Jehl, R. Vitenberg, and H. Meling, "Smartmerge: A new approach to reconfiguration for atomic storage," in *International Symposium on Distributed Computing*, pp. 154–169, Springer, 2015.

[29] N. A. Lynch and A. A. Shvartsman, "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts," *In Proc. of FTCS*, pp. 272–281, 1997.

[30] R. Fan and N. Lynch, "Efficient replication of large data objects," *In Proc. of DISC*, pp. 75–91, 2003.

[31] P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty, "How fast can a distributed atomic read be?," *In Prof. of PODC*, no. 2014, pp. 236–245, 2004.

[32] C. Georgiou, N. C. Nicolaou, and A. A. Shvartsman, "Fault-tolerant semifast implementations of atomic read/write registers," *Journal of Parallel and Distributed Computing*, vol. 69, no. 1, pp. 62–79, 2009.

[33] C. Georgiou, N. C. Nicolaou, and A. A. Shvartsman, "On the robustness of (semi) fast quorum-based implementations of atomic shared memory," in *Distributed Computing* (G. Taubenfeld, ed.), (Berlin, Heidelberg), pp. 289–304, Springer Berlin Heidelberg, 2008.

[34] C. Georgiou, N. Nicolaou, A. C. Russell, and A. A. Shvartsman, "Towards feasible implementations of low-latency multi-writer atomic registers," *Proceedings - 2011 IEEE International Symposium on Network Computing and Applications, NCA 2011*, no. August, pp. 75–82, 2011.

[35] B. Englert, C. Georgiou, P. M. Musial, N. Nicolaou, and A. A. Shvartsman, "On the Efficiency of Atomic Multi-Reader, Multi-Writer Distributed Memory," in *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, (Berlin, Heidelberg), p. 240–254, Springer-Verlag, 2009.

[36] T. Hadjistasi, N. Nicolaou, and A. A. Schwarzmann, "Oh-RAM! One and a Half Round Atomic Memory," *In Proc. of NETYS*, 2016.

[37] A. Anta, T. Hadjistasi, N. Nicolaou, A. Popa, and A. Schwarzmann, "Tractable low-delay atomic memory," *Distributed Computing*, 2020.

[38] C. Georgiou, T. Hadjistasi, N. Nicolaou, and A. A. Schwarzmann, "Implementing three exchange read operations for distributed atomic storage," *Journal of Parallel and Distributed Computing*, vol. 163, pp. 97–113, 2022.

[39] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, "Dynamic atomic storage without consensus," in *Proceedings of the 28th ACM symposium on Principles of distributed computing (PODC '09)*, (New York, NY, USA), pp. 17–25, ACM, 2009.

[40] E. Gafni and D. Malkhi, "Elastic configuration maintenance via a parsimonious speculating snapshot solution," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9363, pp. 140–153, 2015.

[41] N. Nicolaou, V. Cadambe, N. Prakash, A. Trigeorgi, K. M. Konwar, M. Medard, and N. Lynch, "Ares: Adaptive, Reconfigurable, Erasure Coded, Atomic Storage," *ACM Trans. Storage*, 2022. Accepted.

[42] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing Memory Robustly in Message-Passing Systems," *Journal of the ACM (JACM)*, vol. 42, no. 1, pp. 124–142, 1995.

[43] C. Georgiou, S. Kentros, N. Nicolaou, and A. A. Shvartsman, "ANALYZING THE NUMBER OF SLOW READS FOR SEMIFAST ATOMIC READ/WRITE REGISTER IMPLEMENTATIONS," 2009.

[44] P. Kuznetsov and A. Tonkikh, "Asynchronous reconfiguration with byzantine failures," *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 179, no. 27, pp. 1–17, 2020.

[45] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *J. ACM*, vol. 35, no. 2, p. 288–323, 1988.

[46] L. Lamport, "The Part-Time Parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, p. 133–169, 1998.

[47] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of The Society for Industrial and Applied Mathematics*, vol. 8, pp. 300–304, 1960.

[48] L. Jehl and H. Meling, "The case for reconfiguration without consensus: Comparing algorithms for atomic storage," *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 70, no. 31, pp. 31.1–31.17, 2017.

[49] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.

[50] J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal byzantine storage," in *Distributed Computing* (D. Malkhi, ed.), (Berlin, Heidelberg), pp. 311–325, Springer Berlin Heidelberg, 2002.

[51] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi, "Byzantine disk paxos: optimal resilience with byzantine shared memory," *Distributed Computing*, vol. 18, no. 5, pp. 387–408, 2006.

[52] R. Guerraoui and M. Vukolić, "How Fast Can a Very Robust Read Be?," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, (New York, NY, USA), p. 248–257, Association for Computing Machinery, 2006.

[53] M. Bellare and S. K. Miner, "A forward-secure digital signature scheme," in *Advances in Cryptology — CRYPTO' 99* (M. Wiener, ed.), pp. 431–448, Springer Berlin Heidelberg, 1999.

[54] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *The Google File System*, vol. 53, no. 1, pp. 79–81, 2003.

[55] "Colossus." `https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system`, last accessed on 08/10/2022.

[56] "HDFS." `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`, last accessed on 08/10/2022.

[57] "Cassandra." `https://cassandra.apache.org/_/index.html`, last accessed on 08/10/2022.

[58] "Dropbox." `https://www.dropbox.com/`, last accessed on 08/10/2022.

[59] "Redis." `https://redis.io`, last accessed on 08/10/2022.

[60] A. Carpen-amarie, "BlobSeer as a Data-Storage Facility for Clouds: Self-Adaptation, Integration, Evaluation, PhD Thesis, France," 2012.

[61] S. Pan, T. Stavrinos, Y. Zhang, A. Sikaria, P. Zakharov, A. Sharma, S. S. P, M. Shuey, R. Wareing, M. Gangapuram, G. Cao, C. Preseau, P. Singh, K. Patiejunas, J. Tipton, E. Katz-Bassett, and W. Lloyd, "Facebook's tectonic filesystem: Efficiency from exascale," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pp. 217–231, USENIX Association, 2021.

[62] "GFS Reconfiguration." `https://medium.com/@morefree7/google-file-system-notes-baf0c57b9713`, last accessed on 08/10/2022.

[63] Y. Zhang, C. Dragga, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "*-Box: towards reliability and consistency in Dropbox-like file synchronization services," *5th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2013*, pp. 1–5, 2013.

[64] "Redis Topology." `https://blog.devgenius.io/redis-topologies-d9e16a7fa8e0`, last accessed on 08/10/2022.

[65] F. Cristian, ""Basic concepts and issues in fault-tolerant distributed systems"," in *Operating Systems of the 90s and Beyond* (A. Karshmer and J. Nehmer, eds.), (Berlin, Heidelberg), pp. 118–149, Springer Berlin Heidelberg, 1991.

[66] R. Friedman and R. Licher, "Hardening Cassandra Against Byzantine Failures," 2016.

[67] G. Goodson, *Efficient, scalable consistency for highly fault-tolerant storage*. PhD thesis, 2004.

[68] A. Bessani, V. Cogo, M. Correia, P. Costa, M. Pasin, F. da Silva, L. Arantes, O. Marin, P. Sens, and J. Sopena, "Making hadoop mapreduce byzantine fault-tolerant," IEEE Computer Societ, 2010.

[69] P. Costa, M. Pasin, A. N. Bessani, and M. Correia, "Byzantine Fault-Tolerant MapReduce: Faults are Not Just Crashes," in *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pp. 32–39, 2011.

[70] B. Li, M. Wang, Y. Zhao, G. Pu, H. Zhu, and F. Song, "Modeling and Verifying Google File System," vol. 2015, pp. 207–214, 2015.

[71] Y. Kalmukov, M. Marinov, T. Mladenova, and I. Valova, "Analysis and Experimental Study of HDFS Performance," *TEM Journal*, vol. 10, pp. 806–814, 2021.

[72] G. Donvito, G. Marzulli, and D. Diacono, "Testing of several distributed file-systems (HDFS, Ceph and GlusterFS) for supporting the HEP experiments analysis," *Journal of Physics: Conference Series*, vol. 513, no. 4, p. 042014, 2014.

[73] L. Beernaert, P. Gomes, M. Matos, R. Vilaça, and R. Oliveira, "Evaluating Cassandra as a Manager of Large File Sets," in *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, CloudDP '13, (New York, NY, USA), p. 25–30, Association for Computing Machinery, 2013.

[74] H. Attiya, "Robust Simulation of Shared Memory: 20 Years After," *Bulletin of the EATCS*, vol. 100, pp. 99–114, 2010.

[75] A. Anta, C. Georgiou, T. Hadjistasi, E. Stavrakis, and A. Trigeorgi, "Fragmented Object: Boosting Concurrency of Shared Large Objects," *In Proc.of SIROCCO*, pp. 1–18, 2021.