UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Consistency models in modern distributed systems. An approach to Eventual Consistency.

Author: Leticia Pascual Miret
Director: Francesc Daniel Muñoz Escoi
Master's Degree in Parallel and Distributed Computing
09, 2014

Acknowledgments

I would like to thank the enormous patience of my supervisor, Francesc Daniel Muñoz Escoí, and all those people who have been by my side for all these years and for those that are to come. I am not able to mention all of them. It feels fantastic to have a special family where there was no one.

To those who have been standing by my side in the darkest hours.
To those who have believed in me even in the most terrible situation.
To those who are not afraid of living.

Leticia Pascual Miret

Contents

1 Introduction **3**
1.1 CAP Theorem 4
1.2 Consequences of the CAP Theorem 5
1.3 Outline 7

2 Replication **9**
2.1 Introduction 9
2.2 Distributed Systems. An explanation for features and characteristics 10
2.2.1 Synchronisation 11
2.2.2 Fault-Tolerance 13
2.2.3 Consistency in replicated systems 14
2.2.4 Connectivity loss 15
2.2.5 Dependability and scalability 17
2.3 Diffusion Communication Protocols 18
2.3.1 Classification 18
2.4 Replication models 20
2.4.1 Passive replication model 20
2.4.2 Active replication model 21
2.4.3 Semi-passive and semi-active replication models 22
2.5 Conclusions 23

3 Data-centric consistency models **25**
3.1 Introduction 26
3.1.1 Strict consistency 27
3.1.2 Sequential consistency 29
3.1.3 Causal consistency 31
3.1.4 FIFO consistency 32
3.1.5 Cache consistency 33
3.1.6 Processor consistency 34

3.1.7 Other consistency models 35
3.2 Unified Theories of Shared Memory Consistency 36
3.3 Considering implementation aspects 38
3.4 Conclusions 40

4 User-centric consistency models **43**
4.1 Introduction 43
4.2 Session concept 45
4.3 Session or read-write guarantees 46
4.3.1 Read Your Writes 46
4.3.2 Monotonic Reads 47
4.3.3 Writes Follow Reads 47
4.3.4 Monotonic Writes 48
4.4 Conclusions 48

5 Eventual Consistency **51**
5.1 Transparency 52
5.2 Definitions of Eventual Consistency 52
5.2.1 Eventual Consistency as a consistency model 52
5.2.2 Eventual Consistency and a formal definition 57
5.3 Implementation and techniques 57
5.4 Conclusions 58

6 Conclusions **61**

List of Figures

2.1 Passive replication. 20
2.2 Active replication. 22

3.1 Strict consistency model. 28
3.2 Linearisable consistency model. 29
3.3 An example for a sequential execution in a system with this consistency model. 30
3.4 A causal execution in a system with an example of two concurrent writes. 32
3.5 A FIFO or PRAM execution. 33
3.6 A correct cache execution without the blocked processes. . . . 34
3.7 A processor consistent execution. 35
3.8 A processor consistent execution following Ahamad et al. [1993] definition. 36
3.9 The complete lattice of consistency models. 38

Chapter

1 Introduction

"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."
— Lewis Carroll, *Alice in Wonderland*

Eventual Consistency has been considered the master solution to solve the famous *CAP theorem* [Gilbert and Lynch, 2002]. CAP Theorem has been an enlightening summary of what has been told for many years through a lot of researches, that is the impossibility of having a coherent and consistent replicated data in the presence of network partitions in a highly available system. Many publications have been presented and defended about how to implement, what is, and how to define eventual consistency. A common known definition is like the following one: "In the absence of updates, in a quiescent state of the system where all updates have been propagated, all reads will return the same value or all the replicas will share the same state". It is also called *convergence property* as we can deduce from the definition that all the copies of the system will eventually have the same value (or the users will have the perception that there is no modification unless they update some value). However, at the end, how much do we know about its definition, the mechanisms used to achieve it or, more generally, how can we define it formally? These and other aspects of this concept are the main focus of this work. To remark the relevance of being a very desired property present in almost all modern distributed systems that are also called *the cloud*. Due to the performance improvement, there is also a design change in not so big distributed systems, like a web service or a small client-server application.

The main goal of this work is to explain the relevance of eventual consistency in current distributed systems. How the need of extreme scaling without affecting throughput of replicated systems has changed the paradigm of their design, behaviour and implementation. To redefine eventual consistency as the key to a perfect combination for availability, efficiency and scalability.

To this aim, it has been originated a study of the main publications about eventual consistency. Most of the known definitions have been included in this text, comparing differences and similarities among them, leading us to the conclusion that it is not a classical, data or client centric, consistency model.

The evolution of this concept has been born when data replication is used to increase availability and system users, and it took more attention from the distributed system designers since they have seen it as the key solution, combining it with how we let the users perceive the data and reconciliation techniques. In this way, we obtain a system that is scalable and highly available without sacrificing data consistency convergence by weakening consistency conditions. This property, data that converge, seems easily implementable form scratch. Truth is, which kind of consistency level do we need in our system to guarantee convergence? Which mechanisms, techniques and algorithms will help us to this aim? Which consistency models include data convergence? At this point, we can suspect that eventual consistency or data convergence is more about a liveness property of the system than a regular consistency model (concept explained in section 2.2.3, and, in detail in chapter 3). The answer to these questions an more will be found in the following sections and chapters. But before, a little bit from the CAP theorem and its consequences.

1.1 CAP Theorem

Given a distributed system, it is impossible to guarantee the three following properties:

- ***Consistency (C)****: All nodes share the same states, that is, they all have the same data. In an informal manner, a system is consistent if a write is successful, all the components of the system can read the new value.*
- ***Availability (A)****: The system remains operative to take care of every client request, managing it and answering it. Furthermore, it also means that the system is still awake even if a node fails (or crashes).*
- ***Partition Tolerance (P)****: The systems keeps attending client requests even though it has been divided into at least two different parts, also called partition, that cannot communicate between them.*

These three guarantees or properties are desirable for a correct behaviour of distributed applications. What the theorem stands is that it is not possible to achieve or meet these three guarantees at a time in any possible system. Our application can be consistent (C) and available (A) but not tolerant to partitions. Or it can be consistent (C) and tolerant to partitions (P) but not available. Or it can be tolerant to partitions (P) and available (A), but not consistent (C). In any case, one of the three properties must be relaxed. And since the requirements of high availability and high scalability, it is just simpler to sacrifice consistency by being less restrictive about the data access, the data modification and the concurrent access.

This first conjecture was presented in the PODC congress in 2000 by Eric Brewer [Brewer, 2000] and demonstrated by Gilbert and Lynch [2002], but other authors like Davidson et al. [1985] many years before, were describing the same problematic situations and the same solutions to confront them. In particular, in the last citation, two different ways of solving the partition situation in a system are presented. One group of approaches were called pessimistic (to achieve consistency do not let the system to progress) and the others, optimistic or lazy (allow progress no matter what the client sees; eventually, we will be able to merge the partition states and reach a consistent state).

In Terry et al. [1994] this very same situation is solved using what the author called *eventual consistency*: "Practical systems generally desire *eventual consistency* in which servers converge towards identical database copies in the absence of updates. It relies on two properties: total propagation and consistent ordering.". In chapter 5 we will pay more attention to this definition and how it has varied among years and systems. Gray et al. [1996] calls it *convergence property*, inspired from Kawell et al. [1988], who names it as *eventual consistency*. It seems that is the very first time this name is used. We have not found any former publication where the name appears.

If we read for the very first time the CAP theorem, we could misinterpret it and we could reason that is a negative thing that it is impossible to meet the three desirable guarantees in a distributed system. However, in fact, is a valuable consideration to know it in advance since we can manage to design our system considering the limitations that the applications can present and not falling into mistakes or inconveniences that can be predicted in advance and being able to optimize for a particular situation or specific application.

1.2 Consequences of the CAP Theorem

The first obvious consequence of CAP Theorem is that we can only choose two out of three properties or guarantees to be accomplished. The question now is which one should be left out. Well, the answer is also quite trivial:

the one that less affects the regular behaviour that we want to have in the system. However, is not necessary to give up radically to the third. With just some few relaxations to the rule usually is enough. The most common option is to get rid of the C and keep A and P. Thus, we would obtain a BASE system (Basically Available, Soft-state, Eventual consistency) and, if our system works in a transactional environment, some ACID (Atomicity, Consistency, Isolation and Durability) properties disappear. We can say that ACID systems are more focused in keeping a highly consistent system and BASE focuses more in a highly available one. Although, systems like Pnuts [Cooper et al., 2008] prefer to forfeit availability in certain cases. So, the concept of transaction disappears and the concept of session that works with simple operations, *write* and *read*, makes more sense and it becomes the environment that the user needs to communicate with some systems. These strategies among other concepts will be explained in chapter 4.

Under CAP circumstances systems that are wanted to be highly available, highly scalable and cover a wide area have a considerable good option in masking a correct behaviour to the users, considering that users can be people and other applications or services. Henceforth nodes and load can be increased and supported quite successfully. It is trivial to reason that when the higher the number of nodes becomes, the higher the probability of a failure or network partition increases, despite the nodes can be in the same server farm. The most famous systems that appeared and took this option are the *NoSQL databases*.

In conclusion, the situation is depicted as follows. We desire a highly available system which can be highly scalable so it can serve to a growing number of clients or users. The aim is to continue adding nodes while keeping the service working as the load requires, without having too much performance withdrawal. However, data constraints of classical systems make too difficult to manage these new situation and involves a dramatically lowering of system capabilities and the need of synchronisation becomes a heavy task to achieve that increases latency in a non convenient level. Could we imagine a 2PC with thousands of machines implied? The solution to this situation is to relieve all the possible constraints related to data, including synchronisation. The synchronisation is delayed to a near future and the responsibility relies in other parts of the system, like the user application, the broadcast algorithm, reconciliation techniques, new data structures, etc. We can envision with what is said that to programming with these new features will be more complicated.

It is interesting to point out that CAP Theorem is a set of ideas that researchers and system designers had since replication appeared. But the task of Brewer was to give it the relevance that it needed. Knowing this fact, now it is possible to start designing a new distributed system with other scheme in mind having these new features and their implications in mind. Thus, we have a direct consequence of the evident impact in design and, as stated before, in implementation. System nodes will have enough information to make their own decisions without asking for further information to other pairs. There is now also a focus on user behaviour to adapt the application to it and not the other way around. Offline working is allowed in these systems with more relaxed conditions. The application may inform the user about this offline mode or it could be made transparently, depending on the nature of the service, the state can be updated.

1.3 Outline

The text is organised as follows. Firstly, in chapter 2 we will explain some basic concepts of distributed systems, such as replication, consistency, group communication, system interconnection, etc. Always focusing in understanding how consistency arises from replication and the need of a system to be highly scalable. With this knowledge in mind, we will explain in depth the main consistency models and a brief way of implementing them. In chapter 3 we will explain data-centric consistency models plus a formalism to describe them, and in chapter 4 client-centric ones. Both groups are complementary, that is, we can use combinations of various models for a system. After having all consistency models defined, we can envision in chapter 5 what eventual consistency is and which techniques are necessary to implement in order to achieve it. Finally, in chapter 6 a summary will be presented as well as the main reasons why eventual consistency is a desired condition in current distributed systems.

Chapter

2 Replication

"The key to the design of fault-tolerant systems is the provision of redundancy."
— Kopetz and Veríssimo, *Real Time and Dependability Concepts*

2.1 Introduction

The term *redundancy* refers to those extra resources in a system that are not needed in a perfect world ([Kopetz and Veríssimo, 1993]). Replication is a redundancy technique that consists in keeping more than one copy of a component or data in an application. It increases availability and also helps to balance the load between components leading to a better performance. Caching is considered a special form of replication usually, closer to the client side. Nevertheless, it also has a high cost. It is necessary to check periodically the state of each replica, to have a mechanism to incorporate new nodes, to handle probable failure of the existing ones and to propagate the updates to all the replicas while offering a consistent data view and interaction with them.

For all these reasons several protocols must be designed to:

- Manage consistency in a proper manner. This implies update propagation and filtering or blocking of certain accesses.
- Replicas addition to the system. It could need to pause the operations that are taking place in that precise moment. This affects to availability.
- Recovery of lost nodes or partition of the system into two or more subsystems due to network failure. The protocols designed for these aims should not affect to availability, but they could.

We understand a Distributed System as a collection of independent computers or processes that are able to provide a single-system and coherent (consistent) image [Tanenbaum and van Steen, 2007]. A system that achieves these goals, to act like a single machine, is said to be transparent. These not necessary equal computers cooperate like if the system was composed by one single machine, being able to collaborate for a common purpose. As a result, the system has to be able to achieve its goal and be highly available to offer a quality of service for the users. In other words, any component of a Distributed System should be able to stay up to attend client's requests at any moment. A failure should not cause the entire system to go down. In this scenario, it is trivial to think about redundancy as a solution to decrease the probability of the system to stop. Protocols for replication and recovery of the machines will be created and developed to ensure fault tolerance. Scalability, to add more nodes to the system to offer more service quality and quantity, becomes a desirable feature if the applications grow in use rate. All of this, while offering a coherent enough image of the data for the users. With these ingredients we can build a reliable, available and dependable system.

In the first section we will explain these characteristics (synchronisation, fault tolerance, consistency, dependability...) a bit deeper for a better understanding of the work. After that, we will explain the mechanisms needed for the implementation of a replicated system, such as Group Member and Communication Services and protocols for replication and node recovery. Finally, the main four replication models will be described.

In this text, we will use the concept of node as a synonym for machine, computer or process. Definitely, a component of the distributed system. This naming has its origin in graph theory used to represent a distributed system.

2.2 Distributed Systems. An explanation for features and characteristics

In this section we will explain all the main characteristics that arise when defining a Distributed System. First of all, an explanation for the need of a degree of synchronisation. Secondly, and agreeing with Lamport's definition of Distributed systems, we present a small summary of Fault-Tolerance. After that, and since if we replicate the data status, we have written a brief definition of Consistency. This concept will be widely approached through all the text, but a previous definition will clarify our ideas presented here. Following these, we will explain what happens when a node or some subset of the system is isolated from the rest of the components due to problems in the communication. To complete this introductory section, we will explain the concepts of dependability and scalability and why are so important to be considered while designing the system.

2.2.1 Synchronisation

Basically, there are two poles to define synchronisation degree: *synchronous* and *asynchronous*. This notion relates to the time that takes a clock ticking of a node and its partners and, also, to the time that takes a message to be delivered, time of living nodes and the time of their communication. If the system is more strict about this timing by adding boundaries, it will become more synchronous. If the system is less restrictive about it, the system will be closer to an asynchronous one. Depending on what we need in our system, we will design it to be less or more synchronous [Schneider, 1993]. Formally:

Synchronous : A Distributed System is considered *synchronous*, firstly, if there is a maximum in the speed difference for each element of the system (we can think that having this property as something similar to a synchronised clock among all the processes) and, secondly, if there is a bound in transmission time in the communication between any two processes of the system.

An ideal synchronous model would be one where all the nodes share the same clock or all their clocks advance at the same time. This is the maximum level of synchronisation, which is impossible to obtain. In such a system, all algorithms would be very easy to solve and implement since we would be able to know the composition of the system, when any message would arrive to its destination, when a failure would happen (more information about failures will be explained in subsection 2.2.2). Approximations to this model are very difficult to implement and maintain. For instance, a problem hard to face is the addition of extra nodes to the system. It will turn more effortful to synchronise a larger number of nodes for obvious reasons, the size, the latency and the number of the messages required to keep the synchronisation degree will explode with the number of nodes, and the same with the probability of any kind of system failures. Thus, we can start to envision that the growing of the system is incompatible with a strong synchronisation degree.

Asynchronous : A Distributed System is considered *asynchronous* if none of the previous conditions stands for the system. In other words, we do not know precisely if a message will reach its destination or not. The system is considered *partially synchronous* if it attains one of the two properties.

Many problems are not solvable in asynchronous distributed systems since we would not be able to know when messages arrive and, besides, we would not know either how the nodes would progress. For instance, to know the full status of the system, also called snapshot. However, these systems are easier to implement and design. In addition, it is less complicated to add nodes to the system. The more asynchronous the system is, the better scalability (see subsection 2.2.5 for a definition) performance shows. However, synchronisation is not the only factor that affects scalability, as we will see.

Virtual Synchronisation

Other kind of synchronisation approximation is virtual synchronisation. This type is based on the definition of view. It is widely used in high available systems that use membership communication services like Isis ([Birman, 1986]) and Totem ([Amir et al., 1995]). A view is the output of a membership service and it consists of a list of the components or nodes that belong to a given group or system. Each time the configuration of the group changes by adding or removing nodes, the membership service will inform to all the other components with the new list or view of the group and the origin of the change. This is called a *view change*. In this context, we create a virtual time barrier for all the members. All the components of the group are synchronized once the view is delivered and the new configuration of the group, which nodes are up and down, is known. It is important that all the broadcast messages and the notifications of the view changes are carried out in the same order in all the members of the group.

With this condition, the *virtual synchrony property* says: "If processes p and q install the same new view V in the same previous view V', then any message received by p in V' is also received by q in V"'. This property stays that all the processes alive in some view change agree on the set of messages sent and received until that moment. Provided this condition, we are able to manage properly system failures, like crashes, since all the nodes have all the information of all the group and messages delivered. With this scenario is quite easy to achieve a strong level of data consistency (see chapter 3). All the members know who has crashed and who has been added to the group resulting in an easy method to control all the information of the system. Nothing is said about the problems that cause failures but it provides an elegant, simple and correct manner to handle even sequential, causal or FIFO consistencies. Still at all, virtual synchronisation is really heavy to implement highly scalable systems due to the needing of a membership service and the periodical checking of every node in the group. In addition, the requirement of consensus for each change in the system. For a deeper knowledge about virtual synchronisation, the survey Chockler et al. [2001] makes a very good description of it, among other things like membership service and communication.

2.2.2 Fault-Tolerance

From Kopetz and Veríssimo [1993] we take the difference among failure, fault and error. If the system behaves in a different manner that it was supposed to, we say that the system has failed at some point in time. That is, a *failure*. Depending on its characteristics, it can be classified in function of its nature (value or timing), perception (consistent or inconsistent), effect (benign or malign) and oftenness (single or repeated). Lots of these failures are caused by internal incorrect state of the computer, for instance a wrong data or value in a register. This is called an *error* and lasts for a given slot of time. The cause of this error is what we call *fault*. In a similar manner, faults can be classified depending on their nature (chance or intentional), cause (physical or design), boundaries (internal or external), origin (development or operation) and persistence (temporary or permanent).

The goal of a fault-tolerant system is to detect and repair or, at least, mask errors before they show up as failures in the behaviour of the system. Redundancy is the key for such systems. Like replication, to repeat sending the same message until it is received, channel duplication, etc. At the very least, the system has to be able to show the user that nothing important is happening. When a system behaves masking errors to the users we say that it is fault tolerant transparent.

A classification of failures by Schneider [1993]:

- **Failstop**. A processor fails by halting. Once it halts, the processor remains in that state. The fact that a processor has failed is detectable by other processors.
- **Crash**. A processor fails by halting. Once it halts, the processor remains in that state. The fact that a processor has failed may not be detectable by other processors.
- **Crash+Link**. A processor fails by halting. Once it halts, the processor remains in that state. A link fails by losing some messages, but does not delay, duplicate or corrupt messages.
- **Receive Omission**. A processor fails by receiving only a subset of the messages that have been sent to it or by halting and remaining halted.

- **Send Omission**. A processor fails by transmitting only a subset of the messages that it actually attempts to send or by halting and remaining halted.
- **General Omission**. A processor fails by receiving only a subset of the messages that have been sent to it, by transmitting only a subset of the messages that it actually attempts to send, and/or by halting and remaining halted.
- **Byzantine Failures**. A processor fails by exhibiting arbitrary behaviour.

A system designed to handle *failstop* failures is easier to implement than a system that tolerates *byzantine* ones, which is the class hardest to detect and manage.

2.2.3 Consistency in replicated systems

Since there are several copies of the same data, their value can change independently ones from others. In this context appears the concept of consistency. Consistency can be defined as, given a replicated system, how different could be the copies of any replicated value of an object. Thus, a consistency model establishes the access rules to any particular copy and the order of the updates that each model requires. Tanenbaum in Tanenbaum and van Steen [2007], defines a consistency model as a contract between the processes and the datastore. If the processes agree to obey certain rules, the datastore compromises to behave properly. Usually, a process that makes a read operation on a particular data element, expects the return value from the last write operation. Since different copies of the same data are accessible from different processes, it is complicated to keep the last updated value for all of them in order to obtain the image and behaviour of a single machine. In this way, clients cannot perceive that they are accessing to different machines. The idea is similar to an external observer that sees and controls all possible changes in the system so the users do not realise that the system is replicated.

Considering the point of view where we observe the sequence of operation to the data, there are two main types of consistency: server or data centric and client centric approaches. They will be explained in more detail in chapter 3 and 4 respectively. After these chapters, we will explain the causes to the eventual consistency, often called a consistency model. However, in this survey we consider eventual consistency as a condition that is desirable that the majority of the systems accomplish. We will expose all the definitions and arguments for it in chapter 5.

2.2.4 Connectivity loss

In this section we will explain what happens and how is treated a situation of network failures that produces a partition of the network, resulting in the splitting of the system into two or more subgroups. Besides, this partition of the network does not allow to communicate among the subgroups. Now, in this point, what should the system do? If clients continue to access to the service in all the partitions, they could modify in a different and concurrently manner the state of the system and produce as a result that the partitions diverge in their values of the data. Once the healing of the partition takes place, special mechanisms will be needed to recover the convergence of the data and get finally the same state in all replicas. Other approximation is to let accessible only the group that has at least the majority or quorum, i.e. the half plus one nodes, to continue attending clients requests. In this case, no possibilities of divergence of the data can occur, but a part of the clients will be unattended. A drastic solution is to halt the system and reboot it. [Chockler et al., 2001]

The first solution is named ***partitionable***. As it is explained before, the remaining subgroups after the impossibility of communication among them are able to continue serving to the clients. These subgroups are called partitions. This point of view keeps the system fully available meaning that clients continue accessing to the different partitions and can modify data values. However, we can find several complications when the modification of the data is allowed in all the partitions since it is impossible to communicate the changes to the other parts, at least -and in best cases-, for a while. In this situation the update of the state is not shared anymore by all the replicas. Furthermore, once the system is recovered from partition, if each state of the previous partition has diverged, reconciliation techniques must be applied. To solve this incorrect state of the replicas, in Davidson et al. [1985] we can read some proposed techniques for this optimistic strategy, at the end, the system will be fixed and it does not limit availability. To transfer the state or the operations and transactions that took place during the disconnection time. If two or more data modification overlap, then a protocol to decide which result is finally applied. *Thomas write rule* is the easiest solution, the last operation is the one that keeps the result, and the effects of other transactions have to be rolled back (or undone). It can also be applied if we are not working in a transactional environment. Other technique is to run extra operations or transactions that repair the state. *Version Vectors* and *The Optimistic Protocol* are explained in Davidson et al. [1985]. The principal disadvantage is that for some time, clients might be accessing and modifying stale or wrong data. We gain in system availability and, as a consequence the whole system loses the unique image data transparency so data is not consistent nor correct nor coherent.

Partitionable membership services have been used for a variety of applications where a global consistent state is not needed constantly, for example, resource allocation, system management, monitoring, load balancing, highly available servers and collaborative computing applications such as drawing on a shared whiteboard, video and audio conferences, application sharing and even distributed "jam session" over a network. A good list of them can be checked at Chockler et al. [2001].

In contrast, applications that maintain globally consistent shared state, usually avoid inconsistencies by allowing only members of one partition, the ***primary component*** approach [Chockler et al., 2001], to update the shared state at a given time. This is the second solution, to have a quorum or a group with the majority of the system nodes. The remaining subgroups block and are not reachable for users. Thus, creating a decrease of the availability of the system but data consistency is not lost. Once the system is repaired, it is very easy to update the recovered nodes by transmitting the state that has been modified. There are not writing and reading conflicts, there is no need to reconcile any value, just to update it to the last modification. In other approaches of this alternative to partition handling, the nodes from the minor partitions commit "suicide". These techniques developed to maintain the consistency of the system are the pessimistic ones [Davidson et al., 1985]. If no one of the subgroups does not reach the minimum components to become a majority or quorum, the system simply will stop and liveness property will not be guaranteed in the system.

At this point, it is clear that there is a tradeoff between availability and consistency. If we design our system to be tolerant to partitions, the global state may be incorrect for a while during a partition of the underlying network. In the opposite side, if we rather keep our system consistent and correct most of the time, these strong conditions that provide the system a consistent state directly affect progress and availability. We can read this fact back in Davidson et al. [1985] and Chockler et al. [2001]. Later, in 2000, Eric Brewer presented this problem as a conjecture, the *CAP conjecture*. Two years later, Nancy Lynch and Seth Gilbert proved it as a theorem. This is the topic that we have explained in detail in the introduction chapter. This theorem leads to a question very important for the area presented here. What if we want to design a system that has a world wide extension? Such a system implies to have a very wide network with a huge amount of nodes. Therefore, the probability of a partition in the system will become higher. What are we going to do to solve this situation? To stop the minor subgroups leave millions of users unattended it is necessary to give up on strong consistency models and try to research on different approaches to design the systems by analysing and inventing new mechanisms. Such mechanisms must allow the system to progress even in a situation of partition.

2.2.5 Dependability and scalability

In replicated wide area systems Scalability is one of the most important requirements as explained in [Homburg et al., 1996]. In an informal way, scalability can be defined as the ability to support billions of users. This requirement can be interpreted in two different ways. One way is to avoid using components, synchronisation protocols or central algorithms, that present problems for scaling. The other way is to use algorithms or techniques that have a better performance in scaling. However, to scale massively as the Internet requires still is challenging for the design of distributed replicated systems, since the probability of a partition of the system network is increased. Thus, we will have problems with consistency of the data or with the availability, as stated before.

Dependability concepts

The notion of dependability covers the nonfunctional attributes of a computer system that relate to the quality of service a system delivers over an extended period of time. The quality of service relates to the behaviour of a system as seen by a user [Kopetz and Veríssimo, 1993]. The following four measures of dependability attributes are of particular importance:

- **Reliability**: Measure of continuous service delivery, measured by the probability *R(t)* that the system is still providing the correct service after a time interval *t*, if it has been functioning correctly at *t=0*.

- **Safety**: Probability *S(t)* that a system will not fail in a catastrophic failure mode within an interval *t*.

- **Availability**: Measure of correct service delivery with respect to the alternation of correct and incorrect service, measured by the probability *A(t)* that the system is ready to provide the service at the point in time *t*. When the probability has a value greater than 99.999%. In other words, the system is unavailable less than 300 seconds a year.

- **Maintainability**: Measure of the time to restoration from the last experienced failure, measured by the probability *M(t)* that the system is restored at time t if it has failed at time *t=0*.

There is a fifth important attribute of dependability -the *security* attribute- that cannot be measured easily: the ability of a system to prevent unauthorized access or handling information.

2.3 Diffusion Communication Protocols

To work as a single entity, any distributed systems need some communication system to advance in the program execution and to share the information needed to pursue its objective. As a result one of the most important components is the communication protocol that the system uses. Depending on the quantity of receivers and the order of delivering of the messages, it can influence to the availability of the systems and to the consistency. A very good survey that explains these concepts and others related to group communication is Chockler et al. [2001].

2.3.1 Classification

The properties that a communication protocol can guarantee are the following:

Properties and guarantees

- *Atomicity* is the property that says that a message has to be delivered to all or none of the nodes.
- *Reliability* ensures that all the messages will be delivered to all the processes.
- *Order*, all the messages can be delivered in certain order if an order is chosen.
- *Non-duplicity*, a message cannot be delivered more than once to any process.
- *Uniform agreement* if a process has received a message, all the rest of the processes have received it too.
- *Uniform integrity* if a process has received a message if and only if another process has sent it.

Attending to the quantity of the destinies, a diffusion can be classified as a broadcast if we deliver to all the nodes alive, a multicast if it is only a subgroup of the alive nodes or unicast or point to point if there is one sender and one receiver. Typically, broadcast is more used but is hard to implement if the number of nodes of the system is too high. In this particular case, a combination of multicast is used, being the most famous the so called mongering techniques.

We will pay attention to the order of the delivering of the messages in the destiny group. These communication protocols typically guarantee the message delivering to a determined group of receivers. Their complexity depends on the ordering of the messages in the destiny. This complexity can be none, if order in delivering is not needed.

The simplest order in delivering is *FIFO*, the messages are received in the order that each process has sent them. To implement it is very easy: each process adds a number with the identification number and a number indicating the sequence of the message that it has sent. Another typical order is *causal* delivery. Messages are delivered following the causal order defined by Lamport and explained in section 3.1.3. In this case implementation is a bit difficult since it is necessary to carry extra information of other processes in the messages. The most known implementation of this communication protocol is based in logical clocks [Lamport, 1978]. And the last that we comment in this text is *total ordering*, that is, all the messages are delivered in the same order. Is the most hard and expensive to implement, since it needs consensus among the nodes. A very simple way to implement it is by means of using a designated process that is called sequencer. When a process wants to broadcast a message, it sends the message to the sequencer, who will add a sequence number to the message and will make the broadcast to all the destinies. It is very clear that if the sequencer is somehow isolated or crashes, the system stops.

Another approach to communication protocols quite particular is the gossip or epidemic algorithms [Demers et al., 1987]. This technique basically consists in delivering the messages to some of the processes that one particular process is connected. Once a process receives a message, if it did not spread it, then it spreads it to some of the processes that have direct communication. This type of protocols imitates the behaviour of a viral infection spreading and it guarantees that the messages will arrive to all the processes in the system. This kind of communication is interesting in environments where the nodes connected to the system vary rapidly or to interconnect different systems.

Now that we have an idea of how to implement group communication is more suitable to understand why the way the system communicates affects both, availability and consistency. The former is affected if the updates do not reach the destiny and the latter is affected by the order that the messages are delivered. For example, if we have a system with FIFO and reliable communication delivering order, we will have FIFO or PRAM consistency granted. Although this is considered to be true, the equivalence between a communication order and a corresponding consistency model, it has not been already formally demonstrated. This topic will remain as a future work.

2.4 Replication models

In this section we will present four replication models, which are *passive replication* [Budhiraja et al., 1992], *active replication* [Schneider, 1990], and two combinations of both, *semi-passive replication* [Défago et al., 1998] and *semi-active replication* [Powell, 1993].

2.4.1 Passive replication model

In this replication model, also called *Primary backup*, there is one active replica, called the primary server. All the clients send their requests and it processes the requests sending later the answer to the clients and to the other replicas. They are called backups and they receive the result of the processing of any request, as we can see in figure 2.1.

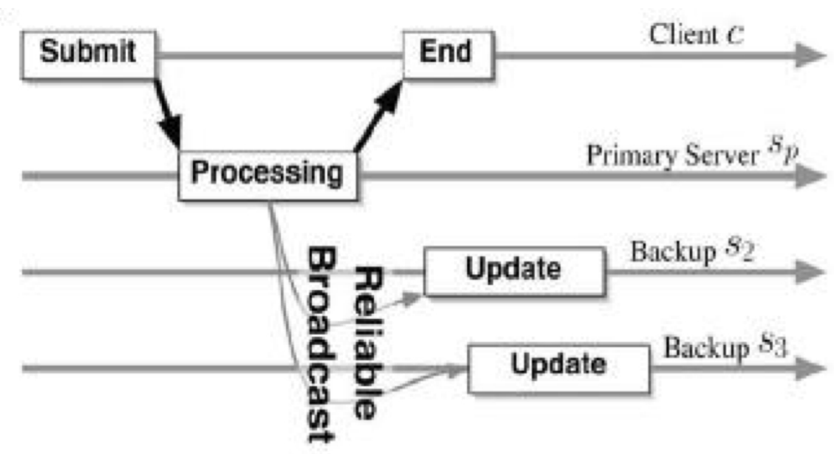

Figure 2.1: Passive replication.

In a relaxed approach of the model, the backups are updated periodically, which relaxes the consistency. In the most strict model, the primary server cannot answer the client until all the backups are updated. This approach is called *pessimistic* because it supposes that the system is prone to fail and some failure may happen and it ensures first that the system is updated before the primary answers to the client; the opposite one is called *optimistic* since it does not consider that the system would fail and first answers to the client and then to the replicas (or at the same time). It is also called *eager* because it focuses on answering fast to the client rather than spread the update to the other replicas; the opposite is called *lazy*, which leaves the "hard work of updating" for a future moment. The principal problem of this replication model occurs when the primary server fails or appears to have failed, it is a single point of failure. In both cases a backup server will substitute the primary, stopping all the system. But, in the second case, a situation of conflict between two primaries has to be solved. Another approach of passive replication permits the backup servers to serve only read requests, allowing an increase of scalability and a soft relaxation of the consistency. Passive replication model is suitable for asynchronous designs, allowing scalability and adaptability or dynamism.

2.4.2 Active replication model

While in the previous model only one single replica has all the responsibility of answering the client and updating the other replicas (which is good in situations of indeterministic behaviour of the code), in this one, active replication, all the replicas receive requests and answer the clients. No protocol for recovery is needed, but if the code that the request executes in the servers has an indeterministic behaviour, problems may arise. It is necessary that all the copies receive the request in sequential order. Otherwise, it could lead to inconsistencies. For this reason, the algorithm in charge of managing a request is more complex than in the passive replication model. It needs a total order reliable broadcast, which implies consensus and, in an asynchronous system, consensus cannot be reached [Fischer et al., 1985]. A basic example of active replication is shown in figure 2.2.

One remarkable advantage of this model is that it supports any kind of fault tolerance, from *fail-stop* to *Byzantine*. In the most severe models, like the latter, we could assume f simultaneous failures if the system is composed by 2f+1 to 3f+1 number of replicas, depending on the communication channels and the synchronisation level. In this case, the client has to obtain a majority of identical answers to understand that the answer is correct. It is convenient to use this model when the operation to execute modifies a quantity of the state of the system is quite large and the time that takes to execute the request is short. In particular, it will be advantageous when the quotient between the quantity of the modified state and the operation

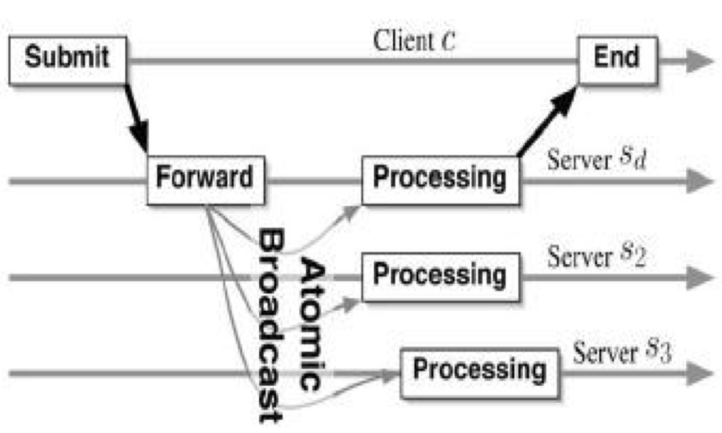

Figure 2.2: Active replication.

execution time is considerable. The bigger the quotient is, the better.

Thus, active model will be suitable to use when the operations executed take less time to execute than the time to transfer the modified state. In the opposite situation, if the operation is longer executing the request than the cost of the transference of the modified data, then the passive model will become a better option. In a scenario where there are some quantity of services to be executed and there are enough nodes in the system to have a primary node for each service then, the system will be faster and more productive with passive replication than using active replication, where tasks have to be executed in a serially. Now, where is the line that separates the preference of using passive or active replication? To answer to this question some practical experiments should be carried out to consider which model suits better in the distributed application to implement. Last, but not least, a couple of important characteristics of the active model are that it supports the worst failures and reconfiguration time is shorter.

2.4.3 Semi-passive and semi-active replication models

Semi-passive replication model is presented in Défago et al. [1998] and it is a variation of the passive model explained before. It tries to solve the situation when the primary fails and it needs to be substituted. All the requests arrive to all the replicas. At this point, we will need consensus to answer the client since there can be an indeterministic answer. This stage is redeemed by using it to propagate updates and to confirm them. In case the primary crashes, the consensus protocol decides which is the next master and it successfully solves if the first primary comes back. The main inconvenient of this model is the needing of consensus makes it very difficult to scale with nodes and operations. It solves gracefully the problem of the master crashing, however this is not a common situation in passive replication. Therefore, it is an interesting replication model from the point of view of theory, but it is not practical.

Finally, *semi-active replication model* tries to overcome the problem of working with indeterministic requests and the filtering made by the client to choose the system answer. Basically, it is the same scheme that active replication, but with a new designated node that acts as a kind of master. Its work is to compute the indeterministic tasks, while the others remain waiting for the output, and to filter the answer of all the replicas to reply to the client. It is also called the *leader-follower* model, it was initially developed for Delta-4 system [Powell, 1993].

2.5 Conclusions

In this chapter we have introduced the main concepts to understand what a distributed system is and how it works. With these notions, the explanation of the following chapters and the challenges that they expose will be more understandable.

Chapter

3 Data-centric consistency models

"The proper order of things is often a mystery to me. You, too?", the Chesire Cat said.
— Lewis Carroll, *Alice in Wonderland*

In this chapter we will explain different consistency models taking the point of view of how data change among replicas. All these definitions are made over a multiprocessor scenario, but they can be applied to replicated environments. That is the reason why along the chapter the words processor or process and multiprocessor are used as synonyms for node and replicated system or group, respectively. They are also called the *data-centric models*. Of course, as we will explain, choosing a consistency model is a tradeoff between increasing concurrency by decreasing ordering constraints and implementation and programming model complexity.

For a better understanding of the examples, we take the formal model and definitions used in Ahamad et al. [1993]. A *system* is a finite set of processors (or processes) that interact via shared memory that consist in a finite set of *locations*. A processor's interaction with the memory is through a series of read and write *operations* on the memory. Each operation acts on some named location and has an associated value. In DSM, the processes interchange messages to share the same memory.

A write operation issued by processor p that stores the value v in the location x is denoted by $w_p(x)v$. Seemingly, a read operation, $r_p(x)v$, returns to process p the value v stored in the location x. A *local history* of a processor p, H_p, is a sequence of memory operations from the execution of a process, including the values read or written by each operation. Now, we can define a *history* as a collection of all the local histories, one for each processor.

An operation is said to be in H if it is in one of the local histories that H comprises. In the definitions of different memory consistency models different operation orders will be used, sometimes in combinations with other orders, to constrain the set of allowable executions.

If o_1 and o_2 are two operations in H_p and o_1 appears first, then we say that

o_1 *precedes* o_2 *in program order* and write $o_1 \mapsto_{op} o_2$. This order is partial on H since it does not relate operations between two different processors.

Different kinds of memories will be defined bellow by considering *serialisations* of certain sets of operations if H is a history, then S is a serialisation of H if S is a linear sequence containing exactly the operations of S. The serialisation is *legal* if each read operation from a location returns the value written by the most recent preceding write to that location, based on the order given by S. If o_1 precedes o_2 in S, we write $o_1 \mapsto_S o_2$.

If H is a history and p is a processor, H_{p+w} contains the subsequences of H consisting of all operations by p and all write operations by other processors. $H|x$ and $H_{p+w}|x$ indicate that the sequences contain operations only of location x.

Here, the memory operations in a process history will be written at the bottom of the example figures, from left to right in program order. Prior to program execution, every location in a shared memory is in some state, the initial value of the location. Similarly, the state after the program has executed is the final value.

A *coherent schedule* is an interleaving of single-address process histories, where every read operation returns the value written by the immediately preceding write operation (except reads that precede the first write, which must return the initial value). The last write in the schedule must write the final value for the location. A multiprocessor execution is considered coherent if a coherent schedule exists for each address. Therefore, all the values that finally each processor sees are the same [Cantin et al., 2005].

3.1 Introduction

As we explained before, a consistency model is essentially a contract between processes, clients or users and the data store and the replicas that store the data. The models that are going to be defined could be consulted in Mosberger [1993] and the citations during the chapter in more detail. Other publication that can be consulted, if interested, is [Steinke and Nutt, 2004], where the author tries to define the consistency using a set of conditions. Each of those sets is built as a list of properties. In the end, the authors show that all consistency models can be defined combining a reduced set of conditions. We will define and explain these conditions in section 3.2.

The models are sorted from the strongest to the weakest. In general, the programming model becomes more restricted (and complicated) as the consistency model becomes weaker.

Strong consistency can be provided by appropriate hardware and/or software mechanisms, but these are typically found to incur considerable penalties, in latency, availability after faults, etc. In every research community where replication has been used, there have been proposals to offer the clients less guarantees that a transparent single-copy image, in order to deliver better performance. For example, strong consistency designs typically require all replicas to receive messages in the same order. A communication system that keeps messages in the same order typically sends all messages through a single sequencer node or on a common bus, and this becomes a bottleneck and a single point of failure for the whole system. If instead we allow messages to be carried independently, and applied to each replica whenever they arrive, performance may be much better, but the clients can see inconsistencies that would never happen with non-replicated data. Those would be weak consistency models.

3.1.1 Strict consistency

Also called *atomic*, it assumes that all the updates are propagated instantly to all the replicas or processors. It is the most strict and synchronous model and it is ideal and impossible to implement, since there is always a latency in communication. The closest approach is to execute the system in a single machine with one single memory. In this approximation, writings are tentative and each time a write is executed, the system blocks all the reading and writing operations. Then, it takes place many decision rounds in order to decide when to apply the change. It is the same case for read operations. Since there is a very strong condition of synchronisation, we will observe that for long periods of time the system is blocked and it is impossible to execute any operation. Thus, the availability decreases one more time, to achieve synchronisation for a strict image of the system, like if we were working with a single machine. In the figure 3.1 below, we can see that every moment that any operation takes place, the time line is separated clearly into time slots that indicate when a value is valid. Each write defines the period of time that a value is correct. This model assumes that the latency is zero and the bandwidth, infinite. Consequently, it is ideal and practically impossible to implement. There is a slightly less strong consistency model that does not block for reading operations. But, for writes the model behaves the same way.

However, Leslie Lamport proposed the *atomic register* concept to specify the consistency model in [Lamport, 1986b] and [Lamport, 1986a]. He defines an atomic register, which is safe (a read not concurrent with a write gets the correct value) and in which reads and writes behave as if they occur in some defined order. In other words, for any execution of the system, there is some

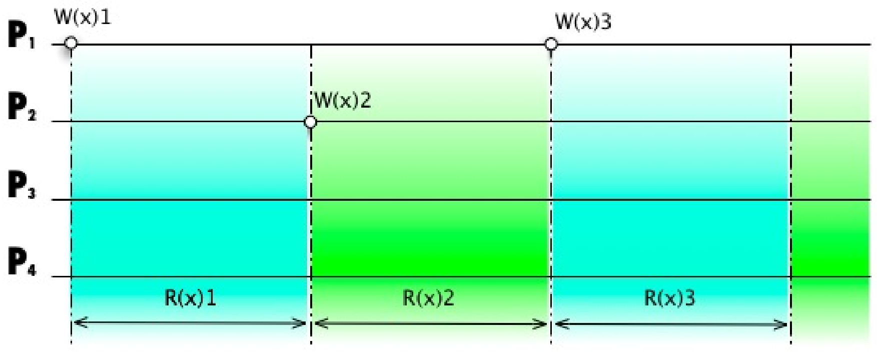

Figure 3.1: Strict consistency model.

way of totally ordering the reads and writes so that the values returned by the reads are the same as if the operations had been performed in that order, with no overlapping (the precise formal condition is developed in the references cited above). An atomic register satisfies the additional requirement that a read is never concurrent with any write. Besides, an atomic register trivially implements one in which all reads and writes are sequentially ordered.

A particular version of this model is the *linearisable model* (figure **??**). Basically, is the same model as strict but it takes into account the delay in the propagations of the data also called latency and it is applied in an environment that processes interact with objects [Herlihy and Wing, 1990]. This consistency model is implemented for objects rather than registers. Linearisability provides the illusion that each operation takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can still be given by pre- and post- conditions. It is also considered a safety property; it states that certain interleaving cannot occur, but makes no guarantees about what must occur.

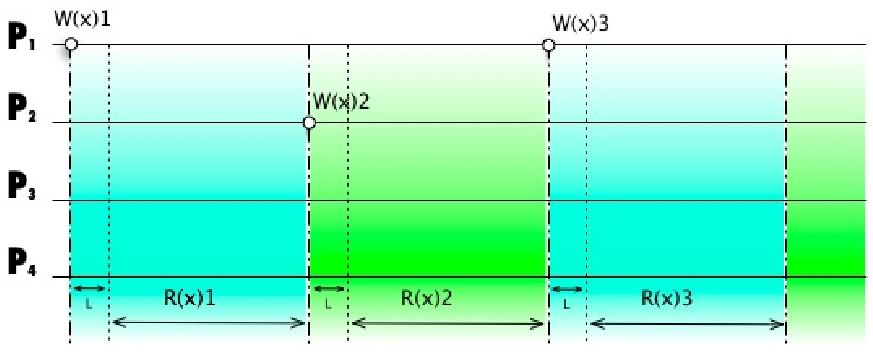

Figure 3.2: Linearisable consistency model.

As a final comment, it is trivial to conclude that these two models are very hard to scale. The conditions of synchronisation and order are so strict that if it is very difficult to implement a system that emulates them very well, to imagine to make them grow with more machines will make the system really slow.

3.1.2 Sequential consistency

This model guarantees that the result of any execution of n processors is the same as if the operations of all processors where executed in some sequential order, and the operation of each individual processor appear in this sequence in the order specified by the program. In other words, sequential consistency [Lamport, 1979] is equivalent to the condition that there is a total ordering on the set of operation executions a condition that can be satisfied even though the events comprising different operation executions are actually concurrent. The atomicity condition traditionally assumed for multiprocess programs is sequential consistency, meaning that the program behaves as if the memory accesses of all processes were interleaved and then executed sequentially. It has been proposed that, when sequential consistency is not provided by the memory system, it can be achieved by a constrained style of programming. Synchronisation commands are added either explicitly by the programmer, or automatically from hints he provides.

Lamport also gave two implementation requirements which, if met, would enforce sequential consistency.

R1. Each processor issues memory requests in the order specified by its program.

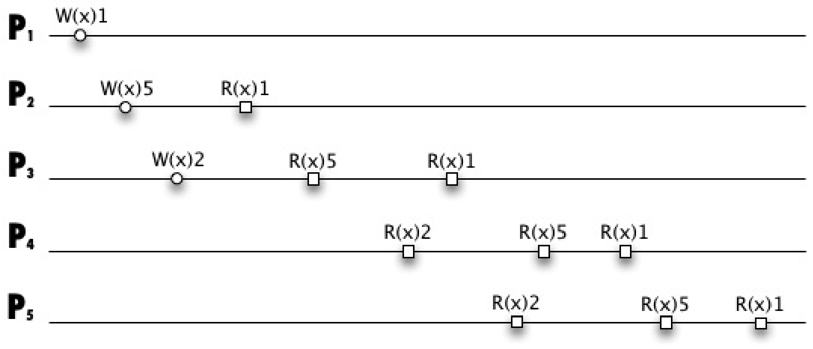

Figure 3.3: An example for a sequential execution in a system with this consistency model.

R2. Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue.

Assuming one writer, in the figure below, we can see an execution where an inversion has happened. A process has read an older value, while other process has read previously the newest value. This problem can never happen in the previous model since it has a severe control of the accesses to the values. This means that if a process reads a new value, no other processes can read any past value. In the sequential model there is not such a control of the time an update is applied and the processes work with different velocities allowing this particular issue to occur.

Now, we can define a *sequentially consistent schedule* as a schedule of the memory operations from an execution that demonstrates sequential consistency is satisfied. In other words, a schedule of all the memory operations, in which every read returns the value written by the immediately preceding write with the same address.

One of the main disadvantages of this model is that makes very difficult to scale since all the copies need to reach commitment in the order of all the updates to apply. But it is a widely used consistency model since it makes all the values of all the replicas to converge to the same value. The easiest way to implement a system with sequential consistency is to use a special process to order the messages by assigning a sequence number to them. All the processes of the system send to it the messages that want to deliver to the others. The sequencer puts the number and then broadcasts it to all the nodes of the system. Then, each one of them will apply the messages in the correct order.

3.1.3 Causal consistency

Causal consistency model is based on the relation *happens before*, the causal order of events defined by Lamport [Lamport, 1978]. It distinguishes between events that are potentially causally related and those that are not. If an event b is causally related with a previous event a then everyone sees the event a before b.

Formally, there are three conditions to define potential causal relation or *happens before*. Given two events a and b, two processes p and q, and a message m, both events are causally related $a \rightarrow b$ if:

1. Events a and b take place in the same process and a precedes b.
2. Message m is sent by p as a send event that we call a. Event b is the reception event in process q.
3. Given a third event c, $a \rightarrow b$ if $a \rightarrow c$ and $c \rightarrow b$. This is called a transitive property.

The writes on the variables are considered as send events and reads are receptions. That is, if there is a message exchange, those set a causal order. Otherwise, it is said that the operations are concurrent and cannot be ordered. It is a decision of the designer of the system or the programmer if it should be an order for those concurrent operations and which one choose to reconcile values.

With this model is no longer necessary the collaboration with other nodes to obtain the correct order (one that accomplishes the model conditions) of the updates to make decisions. So this is a fast consistency model, according to the definition given in Attiya and Friedman [1996]. It is considered for some authors as a weak model since it could allow access to stale data and different replicas may have different final values due to concurrent operations if no reconciliation techniques are applied. In Fekete and Ramamritham [2010] the authors stand that this consistency model is not strong since the replicas do not receive the updates in the same order and stale data could be accessed.

It is weaker than sequential consistency since causal consistency does not guarantee that all the copies converge to the same value, they are not eventually coherent but consistent because it follows the conditions required for the causal consistency model. As we will show later, all the sequential executions are also causal consistent.

In the figure 3.4 we can observe an example of a system that uses a causal consistency model:

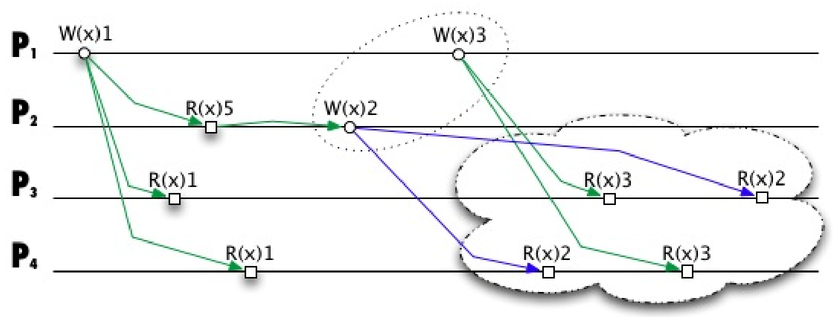

Figure 3.4: A causal execution in a system with an example of two concurrent writes.

3.1.4 FIFO consistency

FIFO consistency, or PRAM consistency [R.J. Lipton, 1988] (demonstration in [Steinke and Nutt, 2004]), says that all processes see all the write operations performed by each process in the very same order that each process has executed them. This condition is called *program order*. So, this model guarantees that any process that reads some data for a period of time, the process will see that all the writes performed by every process will be seen in the order of the program that has been executed. But nothing is said about the order of writes among processes, and they will be observed as a mixture of the program orders of the different processes.

In this weak model, each process has a local copy of the shared memory. To ease scalability, each access has to be independent of the time that takes to access to the rest of other copies. The conditions to accomplish are two:

1. If a processor has a read request, it will return the last updated value of the data in its memory.

2. If it receives a write request, then the processor updates first its memory and, secondly, it broadcasts this new value to the rest of the processors.

In the figure 3.5, there is an example of a FIFO or PRAM execution, and we can see that Each processor will see its writes in FIFO order and the writes of the other processors may be perceived in different interleaving in each process.

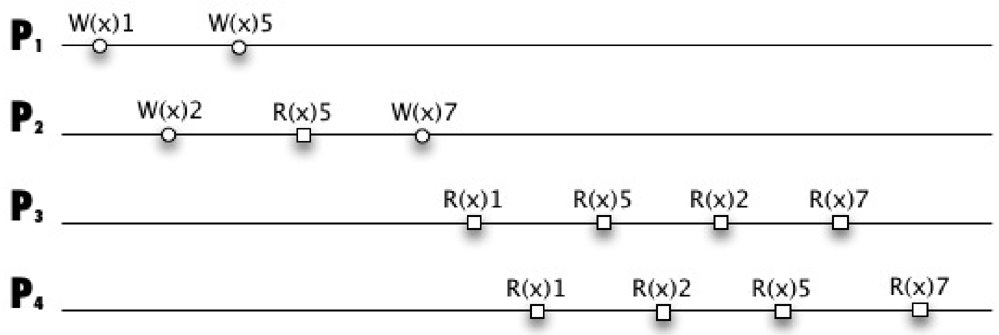

Figure 3.5: A FIFO or PRAM execution.

3.1.5 Cache consistency

Cache consistency model has been firstly defined by Goodman [1989]: *"each read of a memory location is guaranteed to obtain the most recent value".* In DSM, we change memory location by variable. Thus, writes to different memory locations, or variables, may be observed to occur in different order by different processes. It is similar to a FIFO access, but considering the accesses to a data variable instead of the operations of the program order, as figure 3.6 shows. This is quite an interesting model because every variable will be read with a correct sequential value, but the global sequence seen by all the processes may not be sequential since concurrent access to variables will alter the program order. Thus, we can observe sequences with accesses to the variables that do not correspond with the order the writes were issued by each process or processor. As a result, we will see always the last value written in the variables and the vision of all the replicas of the data will be coherent or convergent.

Later, in Gharachorloo et al. [1990] this consistency model is also called *coherent consistency* for this peculiarity. All the values of the replicated variables are convergent to the last updates but without being as strict as sequential consistency. The restrictions that the program order imposes are partially removed. We may observe the program order but we perceive some sequential order in which any variable is accessed and always the last written value. This order may only respect the program order for the variables independently. This situation can cause that the processor could have executed the write operations to different variables in any other order totally different that has been seen by the reader. In case that we have one only variable with control of concurrent access in our system, we will see that the sequence observed will follow also PRAM consistency, but it is a very particular, and the only one case that a sequence can guarantee both models. Another difference between those models is that while PRAM consistency does not need the collaboration or agreement with other processes to establish the sequences that the others will observe, to implement cache consistency will be needed to inform the processes which is the order that has to be perceived.

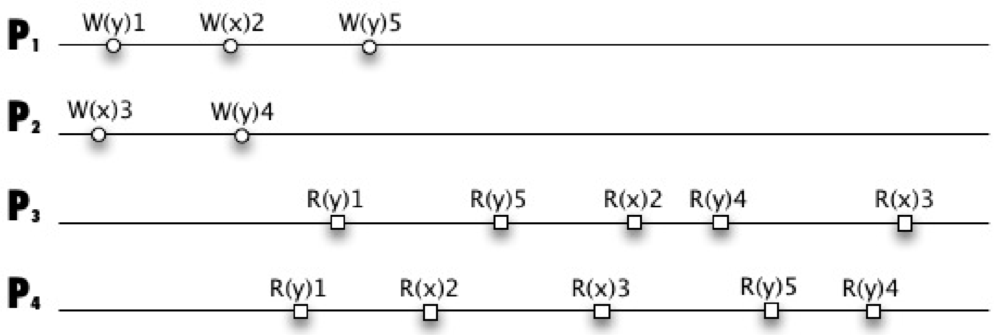

Figure 3.6: A correct cache execution without the blocked processes.

It is considered the weakest consistency model in a multiprocessor environment. To implement it in a DSM system will be quite complicated and some nodes have to be blocked in order to preserve the sequential access to the variables. As a consequence, the system performance will decrease considerable. Therefore, it will be very complicated and practically inviable to scale up while keeping this consistency model. In such conditions, to implement a sequential model would be as expensive as implementing cache and the system will accomplish more data guarantees, like convergence of data and program order. See figure 3.6 for an example of a sequence of cache model consistency behaviour.

3.1.6 Processor consistency

The *Processor consistency* model is the combination of PRAM consistency plus Cache consistency, which means that is strictly stronger than both. Both consistencies, PRAM and Cache, have been explained before for a better understanding of the model presented here. Processor consistency was firstly presented by Goodman in Goodman [1989] and later, in Ahamad et al. [1993] is defined formally and a stronger version of it is proposed.

Goodman's first definition says as follows: "A multiprocessor is said to be *processor consistent* if the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program ". At first sight, it might be very similar to PRAM consistency. However, Goodman considered in the cited article that this consistency model lays in the middle of a line where strong or sequential consistency and weak or cache consistency are the maximum and the minimum respectively, guaranteeing then the conditions of cache consistency. Thus, Processor consistency is the combination of Cache and PRAM consistencies. An example sequence of Goodman's definition in figure 3.7.

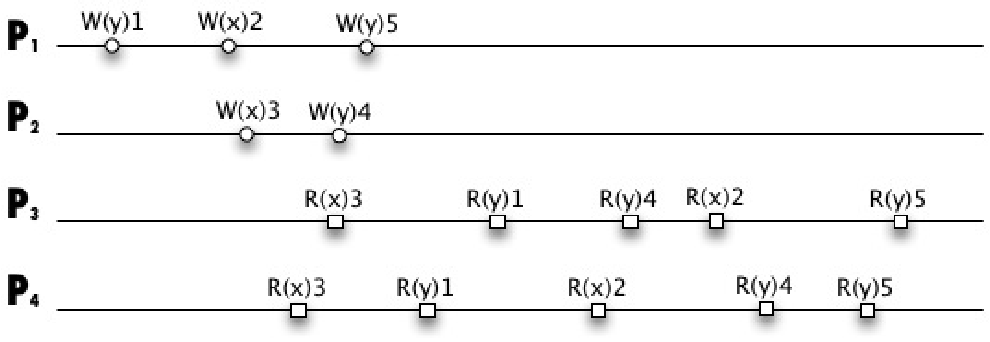

Figure 3.7: A processor consistent execution.

Due to different interpretations of Goodman's definition of processor consistency, in Ahamad et al. [1993] claims that in Gharachorloo et al. [1990] there is a redefinition of it to avoid sequences of operations not desired. It combines coherence with "semicausality", which is very similar to the second and the third conditions of the formal definition for causal consistency (see section 3.1.3). This change is defined by the weakening of writing process order and reading process order, in other words, a weaker notion of the program order. Reads are allowed to "bypass" earlier writes by the same processor. We could say that the different definition for Processor consistency is the addition of causality among processes.

Ahamad himself says that a system that implements any of these two definitions for Processor consistency will have problems maintaining and implementing the necessary mechanisms for a correct behaviour to accomplish the guarantees that this model offers. In this case, he arguments that presented these limitations, it will be easier to implement a sequential consistency model that is more strict.

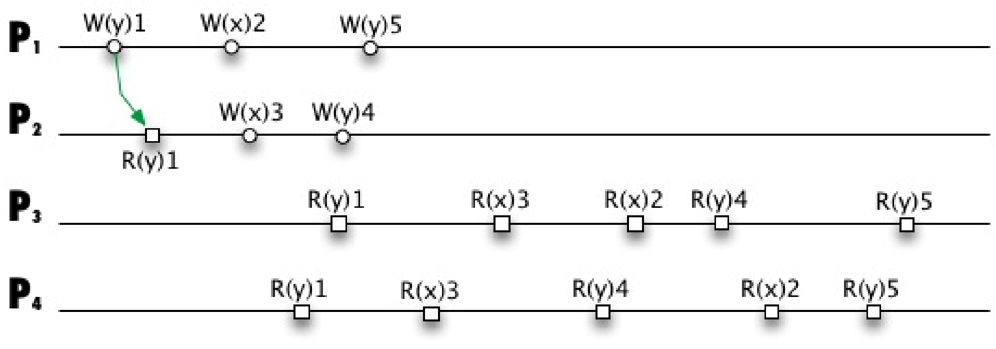

Figure 3.8: A processor consistent execution following Ahamad et al. [1993] definition.

3.1.7 Other consistency models

We have described the consistency models that are related to the purpose of this text. But there are others, some weaker than the ones presented here, like *Slow memory*, where the write operations are propagated very slowly [Hutto and Ahamad, 1990] or *Local consistency*. The later one is considered the weakest consistency model of all. Many others are related with synchronisation.

A multiprocessor is *slow consistent* if reads must return some value that has been previously written to the location being read. Once a value has been read, no earlier writes to that location (by the processor that wrote the value read) can be returned. Writes by a process must be immediately visible to itself.

3.2 Unified Theories of Shared Memory Consistency

Several classifications of memory consistency model have been developed. From all of them, we choose the classification presented in Steinke and Nutt [2004] for its expressionism of qualities and conditions, that defines a consistency model as a combination of one or certain set of these conditions. The other classifications organise memory models as being stronger than others or being contained in others [Cholvi and Bernabéu, 2004], but they not insist upon defining them as a set of conditions used to prepare a consistency model, like a food recipe. Steinke and Nutt allows us to reason with these properties more easily and visually decompose any consistency model into properties. From the properties we can infer new consistency models, not knowing if they are convenient for real systems, but they can inspire or introduce new approaches to implement and consider if it is implementable and suitable for a real case.

GPO - Global Process Order The condition that there is global agreement on the order of operations from each process.

GDO - Global Data Order The condition that there is global agreement on the order of operations to each variable.

GWO - Global Write-read-write Order The condition that there is global agreement on the order of potentially causally related writes.

GAO - Global Anti Order The condition that there is global agreement on the order of any two writes when a process can prove it has read one before the other.

One could do some associations with these properties and other ones defined in other consistency models, such as the *semicausality* condition [Ahamad et al., 1993] is very similar, if not the same, as GWO. A consistency model is said to be stronger than another if every condition required by the weaker model is also required by the stronger one. Thus, a stronger consistency model has a more highly constrained behaviour than a weaker one. Sequential consistency is strictly stronger than processor consistency which is strictly stronger than cache and PRAM (FIFO) consistency. However, PRAM and cache are not comparable between them as well as with causal. At the same time, causal consistency is stronger than PRAM. From up to down, the consistency models are ordered from stronger to weaker. Those that are related with the relation stronger than (or weaker than if we look the opposite way) are indicated with a line. Those that cannot be compared are not connected by any line.

From the bottom to the top, the models are described by the combinations of the properties. Starting from the weakest, the bottom, we found local consistency, where no conditions or properties have to be accomplish. Directly stronger we found immediately slow consistency, defined by a relaxed version of a combination of GPO and GDO. In this particular case, two or more properties can be combined to obtain a weaker model. We could combine more properties to find weaker models than the properties that define the consistency models. In the other cases, in this example, we will see that the combination of properties will result in a stronger consistency model. Following the left branch, we find FIFO or PRAM consistency model,

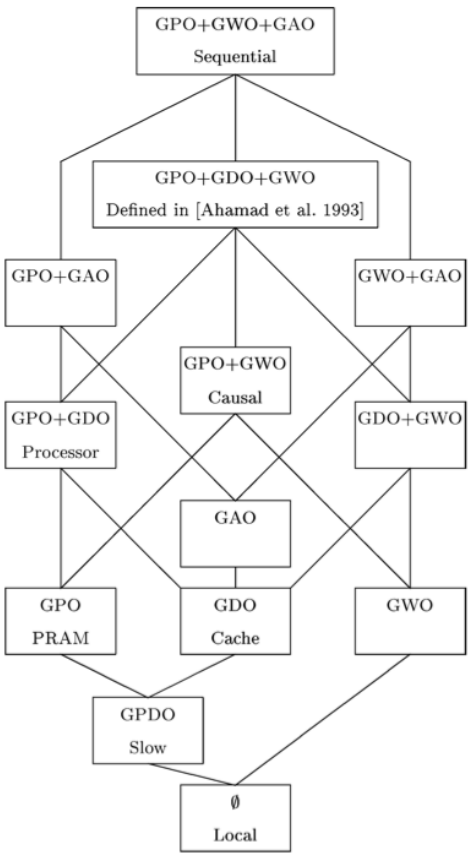

Figure 3.9: The complete lattice of consistency models.

defined by GPO. If we add GDO to this property, we obtain processor consistency defined by Goodman [1989]. Strictly stronger becomes the model if we change GDO by GAO. Going back to slow consistency and taking the right branch, we find cache or coherent consistency characterised by GDO. Strictly stronger there is GAO property, which defines a consistency model not named yet. Now, returning to the bottom of the lattice, and taking the right branch, semicausality property or GWO is stronger. If we add GPO or process order, we obtain causal consistency. Other new consistency models can be inferred by adding GDO and then GAO for a stronger one. The authors suggest that those models without names could be interesting to be explored. Above all the combinations that includes a subset of GDO, GWO and GPO we find processor consistency defined by Ahamad et al. [1993]. Finally, the strongest one, at the top, there is sequential consistency, formed by the properties of GPO, GAO and GWO.

3.3 Considering implementation aspects

In this section some basic techniques to implement the main consistency models explained in this chapter will be seen briefly. Of course, there are many possible implementations apart from these presented here, which are some of the most familiar among distributed systems.

Sequential consistency can be achieved by means of a FIFO total order broadcast. A simple idea to obtain this broadcast is implementing a sequencer. It is a designated node of the system that receives the messages to send to all the nodes and adds them a sequence number. Then, the rest of the nodes and itself will receive the messages with this sequence number that helps them to sort the messages to be applied. On the negative side of this implementation we have that the designated node can crash and then the system must have a mechanism to detect it and then to choose another node as the sequencer. The reader will notice the similarities with passive replication model and the primary crash problem, indeed, it is practically automatic to implement sequential consistency in a pure passive replicated system. Another typical implementation is using a token. Furthermore, one can trivially realise that these methods are hard to scale provided that in the first example the sequencer is a bottleneck and in the second case it could take a really long time before a node gets the token. The node that possesses the token has the power of adding the sequence number to the message and to broadcast it to all the rest of the nodes.

Next consistency model, causal, can be implemented directly with a causal broadcast. Moreover, it can also be resolved with a reliable FIFO broadcast plus additional causal information needed in the messages. This was not necessary in the first case because the communication protocol gives the support for causality information and in the second case is mandatory to include it at the programming level and making it more difficult, alleviating the mechanisms to support the communication protocol. This second approach is beneficial for system composition, since it is easier to implement FIFO channels to interconnect several systems allowing an open door to a seriously big scalable behaviour.

As well as causal consistency, FIFO or PRAM consistency model can be implemented with the same communication protocol and, in particular, can be implemented automatically with just a FIFO broadcast. Evenly, with a reliable broadcast will be enough if each process or node of the system adds a tuple with its identifier node number and a sequence number of the operation in it. If we examine this information before delivering (understanding delivering as the process of analysing the message content after it is received) the message to the application. Again, by simplifying the communication protocol, the software becomes a bit tricker.

Furthermore, these two models, causal and PRAM, are considered fast models, following the definition of fast consistency models from Attiya and Friedman [1996]: "An implementation, satisfying a certain consistency condition, is *fast* if the execution time operation is significantly faster than the network delay". Besides, they do not need any kind of consensus or collaboration (meaning and implying synchronisation) with other nodes of the system. This characteristic, apart from allowing them to scale up easily, provides to the system to be consistent when the system reconnects and heals from a network partition. The data probably will not be the same in all the replicas, but the conditions to accomplish the contract of the consistency model are not violated. These two considered weak consistency models solve the CAP Theorem described in the introduction of the work. Although this may be incredibly useful, there is the problem that the data is not coherent, and usually coherence of the data is desirable in a system. We have finally arrived to the necessity of mechanisms to make these models to converge, to make them eventually consistent. *Eventual consistency* is a trending topic in late years in distributed systems due to the use of these weak consistency models to implement high scalable systems. It will be discussed and explained deeply in chapter 5.

Cache consistency, where the access to the variables is sequential, the implementation proposed is similar to the approach of the sequencer for sequential consistency, but with as many sequencers as variables. Then, with a FIFO broadcast will be enough. Thus, the problem with this model is the same as the sequential one, even worse, since there are as many sequencers as variables, which makes the system really hard to work and scale. Cache consistency model makes sense in a multiprocessor machine since the access to the registers already provides this consistency, as we can read in Goodman [1989].

Finally, an example to implement processor consistency would be the combination of a FIFO and cache implementation, sequencers for the variables and a FIFO communication protocol. One more time, we will have the same problems than with cache consistency.

3.4 Conclusions

In this chapter we have explained a consistency model considering the point of view of data and ordering of operations. A consistency model is defined as a set of rules, or restrictions, to accomplish. It is a contract The models presented here were strict, linearisable, sequential, causal, processor, PRAM or FIFO, cache or coherent, and the mention of other consistency models. From this group, the first two are the only ones that requires of explicit synchronisation, while the others are designed in a manner that is possible to lower the degree of synchronisation by making accessible that should be already updated in the strict model. Moreover, reorganisation of messages before updates being executed, undoing and redoing of message effect as well are allowed. This fact tell us that synchronisation plays an important role in the consistency of the system. Moreover, the more synchronised the systems needs to be in order to achieve consistency rules and, what is more important for modern distributed systems, it complicates scalability and, thus, partition tolerance. Going back to the consistency models explained, those that did not need additional mechanism to allow the data to converge are complicated to scale to a large number of nodes and keep consistency with a reasonable system throughput. This is an important reason why systems are being implemented nowadays using asynchronous programming.

It is interesting to point out that, even considered as a weak model, cache consistency, defined by the GDO property, is considered a convergent model since all data values remain coherent. These model and those that have as an ingredient the GDO (or GAO, that is strictly stronger) will be data convergent. However, the models that do scale well are those that are not data convergent, they do not accomplish GAO or GDO and additional mechanisms will be needed to this end. Therefore, convergence is a property that some consistency models have and is interesting for the majority of the distributed applications. Such a concept is called a consistency model, *eventual consistency*, which will be explain in detail in chapter 5.

All these reasoning lead us to an interesting question. If we want a system to be scalable, we have to choose a consistency model that does not converge naturally and we have to implement some techniques to solve conflicts and data incoherency. What do we do while these problems are fixed? In the following chapter we will see client or user centric consistency models to start answering this and other questions.

Chapter

4 User-centric consistency models

"Life is what happens to us while we are making other plans."

— Allen Saunders

4.1 Introduction

Weakly consistent systems rely typically on the proper handling of some periods of time were inconsistencies may appear. The most common cause for inconsistency is the latency or the time that a correct update takes to arrive to all the copies. Node failures and network partition are less common but more dangerous since they derive in more complicated situations to solve, if there is a solution. Users may be aware of this characteristic of the system and may be able to live with it knowing that information will be repaired soon enough. Or it can be more preferable that users are never aware that incorrect behaviours indeed happen by masking them. Convergence among copies is a guarantee that clients should count on and is desirable that a system could provide it. Of course, normally, in these systems updates are not indivisible and atomic transactions but just simple operations to change, create or delete a value. This conception of a weakly consistent system is given already in Oppen and Dalal [1983]. In this case, users are aware that they will face with inconsistencies. But, what if there is a manner to hide these incorrect behaviours and have the appearance that everything is just working fine? Client-centric consistency models care about the client perspective of the data, allowing the system consistency to be weaker and sometimes not noticed. We can consider then two main kinds of consistency in the system: the one perceived by the user and the one that actually the system is working with. Of course, other consistency levels in an application can be considered and exploited [Alvaro et al., 2013].

Up to this point, we have explained how to understand and define consistency looking at the data changing in the server copies. But, what about the clients or users that are connected to the system? How do they perceive these dynamics in the data that they are accessing and, significantly less commonly, modifying? This is the main idea of the user-centric consistency models, just to focus on what the clients or users perceive from the data they are managing [Terry et al., 1994]. Clients or users are allowed to access any replica by means of a session and, additionally, to accomplish some combination of certain guarantees, the users will see -or think that they see- that the application meets the consistency condition expected. The *session* concept and the *session guarantees* will be explained in the following sections.

As we have seen in the previous chapter, stronger and naturally convergent consistency models are more difficult to implement and heavier to maintain due to synchronisation and the need of consensus and certain restrictions in the order of operations. On behalf of not allowing the user to see what is really happening in the system, we could use some tricks to make inconsistencies transparent to the user. Under those circumstances, if we are able to make the user perceive or think that the system performs completely well and without any problem like failures, inconsistency periods, network partitions, upgradings, migration or whatever any other possible inconvenience would be a huge benefit. Consequently to use certain techniques to mask inconveniences will not be a bad idea. They will allow to relax the system consistency model, and hence get some benefits of weaker consistency model management, making possible to solve these situations without giving the impression that the system is indeed misbehaving or making something different to serve the clients.

What is really interesting is that once these techniques are defined, it is possible to use a more relaxed consistency model in the system. Therefore, we can use a weaker consistency model such as FIFO or causal in the replicas and the user will perceive that they are seeing things and system behaviour like if they were working with a stronger convergent one. Of course, not only extra mechanisms will be added for recovery and convergence of the data replicas, but also availability or perceived consistency will be compromised sometimes. The relaxed mechanisms typically comprise a gossip or mongering broadcast algorithm to spread the updates, to remove the complexity of managing transactions and use instead simple operations -write and read-, to keep versions of the values as well as certain order in writings, reconciliation values techniques (Thomas Write Rule could be the simplest one combined with version values). These mentioned mechanisms shall not interfere nor affect session guarantees and neither different sessions among them.

The chapter is divided as follows. First of all, we will explain the principal conception that will grant that the guarantees explained in the following sections may take place. Secondly, we will explain four session guarantees.

Finally, we will show the principal implications that these concepts have in implementing applications. It is important to highlight that the guarantees and concepts explained below are complementary to the guarantees that a regular or data-centric consistency model offers. Perhaps, in some cases, small modifications to the original system could be needed, but minor ones.

4.2 Session concept

This concept is the fundamentals to understand and implement the user-centric consistency models. A *session* is an abstraction for the sequence of read and write operations performed during the execution of an application. It starts once the client connects to one server and it ends, idealistically, when the client has finished all the operations needed. All the interaction has been logically performed with the same server during all the session. In this way clients only see that their actions have a consistent enough behaviour in the system, even if there are other clients accessing concurrently to the same data or if they read and write from very likely to be inconsistent replicas. Within the environment of a session the guarantees that are explained later can take place. These are:

- *Read Your Writes*, read operations reflect previous writes.
- *Monotonic Reads*, successive reads reflect a non-decreasing set of writes.
- *Writes Follows Reads*, writes are propagated after reads on which they depend.
- *Monotonic Writes*, writes are propagated after writes that logically precede them.

Of course it is important to repeat that each Read or Write that a user does goes directly to the same server under the session, but it can happen that a connection could be lost due to any kind of network problem. In these scenarios is convenient to retry the user reconnection to the same server, if available. In the case the server is unreachable, then the next action is to connect the user to a server that has the same state. Otherwise, the user should start again all or part of the work with the new server. Thereupon the client will realise that some failures have happened. The task of providing the guarantees we just mention relies primarily on the *session manager* through which all the reads and writes operations are serialised. It can be considered part of the client stub that communicates with the replicas. For each session it maintains two sets of write identifiers:

- *read-set*, is a set of write identifiers for the writes that are relevant for the session reads.
- *write-set*, is a set of write identifiers for those writes performed in the session.

We do not have to confuse sessions with transactions. Sessions do not ensure atomicity nor serialisability. The real intention is to give the image of a central database to the user, even if the client is working with different and potentially inconsistent servers.

4.3 Session or read-write guarantees

In this part of the text we are going to describe in more detail the session guarantees mentioned in the previous section. Of course, in a system using client caching these guarantees can be applied. We have to consider that depending on what the clients need to see in order to keep an appearance of a strong consistent system while, in fact, the replica consistency is weak indeed. Sometimes it will be necessary to make a combination of a couple or more of them or it works just fine meeting only one guarantee. This is possible thanks to the scope that the session provides to the communication, i.e. writes and reads, between the server and the client. Considering DB(S,t) as the ordered sequence of Writes that have been received by server S at or before time t, the properties that can be assured in a session environment are enunciated below.

4.3.1 Read Your Writes

Read Your Writes property arises from the fact that natural order of operations for a user is to be able to read what they write in the very same sequence. As long as the session lasts, clients will read the last value they have updated. Specifically:

RYW guarantee *If Read R follows Write W in a session and R is performed at server S at time t, then W is included in DB(S,t).*

Considering applications, reads within the session may see other writes that are performed outside the session, since there may be other users performing writes to other servers and these updates could be seen in the present session and server. Thus, a user might perceive the interaction with the system by other users. To sum up, any user will see her writes in the same temporal order that the user has executed them and can also see other user interactions with the system.

4.3.2 Monotonic Reads

This guarantee tells us about the updating process of the system as it is accessed by writes. The users can see the evolution of the system as they read through time increasingly within the session. MR produces the impression to the users that the system is growing, changing and working through time and it does not go back to the past by showing stale or expired values for their particular session. A write-set is said to be *complete* if it is the minimum sequence of ordered writes that produces a given R when we perform R in DB(S,t). Assuming that, the function *RelevantWrites(S,t,R)* returns the smaller and unique write-set sequence that is complete for R. Now, we can define formally MR as follows:

MR guarantee *If Read R1 occurs before R2 in a session and R1 accesses server S1 at time t1 and R2 accesses server S2 at time t2, then RelevantWrites(S1,t1,R1) is a subset of DB(S2,t2).*

That clearly tell us that the DB changes and grows through time and it does not go back in time by showing past values. One interesting thing about this guarantee is that if we design the client stub to achieve it, in a system where the consistency data model is sequential will avoid and solve the problem of inversions.

4.3.3 Writes Follow Reads

The Writes Follow Reads guarantee ensures that traditional write/read order dependencies are preserved in the ordering of writes at all servers. This means that in every copy, writes made during the session are ordered after any writes whose effects were seen by previous reads in the session.

WFR guarantee *If Read R1 precedes Write W2 in a session and R1 is performed at server S1 at time t1, then, for any server S2, if W2 is in DB(S2) then any W1 in RelevantWrites(S1,t1,R1) is also in DB(S2) and WriteOrder(W1,W2).*

At first, it looks familiar to causal consistency, but from the user point of view. Besides, it also implies to see the write operations from other users outside the session. The user of a particular session can observe the writes of other users with their own sessions.

This guarantee can be relaxed into two constraints in write operations. A constraint on write order that ensures that a write properly follows previous relevant writes in the global ordering that all replicas will eventually reflect.

And another one on propagation that ensures that all servers (and clients) only see a write after they have seen all the previous writes on which it depends. Formally:

WFRO *If Read R1 precedes Write W2 in a session and R1 is performed at server S1 at time t1, then WriteOrder(W1,W2) for any W1 in RelevantWrites(S1,t1,R1).*

WFRP *If Read R1 precedes Write W2 in a session and R1 is performed at server S1 at time t1, then, for any Server S2, if W2 is in DB(S2) then any W1 in RelevantWrites(S1,t1,R1) is also in DB(S2).*

4.3.4 Monotonic Writes

This guarantee says that a write is only updated in the server if the copy includes all the previous session writes. Also, it implies the user of the current session and other users from outside the session.

MW guarantee *If Write W1 precedes Write W2 in a session, then, for any server S2, if W2 in DB(S2) then W1 is also in DB(S2) and WriteOrder(W1,W2).*

Thanks to this guarantee, the user session can be sure of the order of two given writes. It also affects to users outside the session. Obviously, this guarantee is harder to keep since involves coordination among replicas and users to agree in a write order. In weak systems, write conflict orders can be resolved after the users may perceived them. It is a decision of the programmer to allow clients to see these temporal inconsistencies. The more the programmer wants the users not to realise that something is wrong, the more difficult will be to find servers that are enough up to date to be accessed. Thus, affecting availability. One more time we see that to keep strong guarantees about consistency implies a degradation in system availability.

4.4 Conclusions

The concept of session and four guarantees have been presented in this chapter, among the description of an example of a system that implements a weak data consistency model, causal consistency in this case. Furthermore, the mechanisms applied to spread the updates are the so called mongering or gossip, which guarantees that all the replicas will eventually receive all the updates. Thus, contributing to the system convergence. However, some extra mechanisms are added to the system to solve data conflicts, but they can be implemented using the communication mechanism or just with a simple solution like version vectors. One could develop extra guarantees or modifications of the presented ones in convenience of the application or usability or the guarantees that we wish to offer. As it has been said before, they can be used to help to mask other kind of failures in weak and stronger consistent systems.

On the one hand, what we are obtaining with this approximation is a more relaxed consistency mechanism for the whole system, since the user is not aware of what is happening in the servers. On the other hand, it will complicate the implementation for the programmers and it will exist the problem that users could realise some inconsistencies. Despite of these withdrawals, at the end, a solution will be found for it, if the application really needs to care about it.

As we have seen the solution to give the appearance of a central and unique image from the user point of view, we have realised that if we wanted more similar to sequential consistency, more difficult was to implement it and less servers or copies we could access to have this look. There is, in fact, a trade off between availability and consistency as well at client side. However, at least, we are able to manage more perfectly, more transparently weak consistency models from the data point of view, like fifo or causal.

It helps to complete the consistency models to achieve the global apparent consistency in the system, that is, the consistency level that we want the users to see or think that they are working with. Besides, with these guarantees is possible to solve or alleviate inconsistencies or consequences of write conflicts with models that not imply a global consensus decision for the operation order. It is possible to fake it somehow by combining the consistency model applied in data replicas and some of the guarantees explained in this chapter. The author proposes that more guarantees can be defined as it may arise from the application used in combination with the data file replicated system used in Terry et al. [1994]. At this moment, the guarantees are presented to work with a system that clients are mobile and can be offline most of the time. Nowadays, with the huge growing in numbers of nodes and users of the Internet, it is common sense to use them for improving the scalability and availability of the system by relaxing data consistency model.

If we delay the replicated data updates, the techniques to keep data consistent will have less influence in availability and obtain better performance and higher scalability. We do not need take into account a strict coordination (synchronisation) among replicas in order to have in all of them the same value. In this scenario it may happen that a user reads an old value for a variable after writing a new value. A simple solution is just to attach a client or user to one server or replica by means of a session. Other thing that is interesting in the user-centric consistency model is that we can combine some of the four guarantees depending on what the client would tolerate as an inconsistency in what it sees. For some applications is not really important if the client sees a non correct order of updates, or if the result changes. We can think about an e-mail client. We can connect to some server, to erase some emails. Then, connect from another mobile device and not perceive the changes we have made in the previous session. It's a matter of time, if the designer decides so, that it is not important to what replica we are asking for information. What it matters is that eventually we will be able to see the correct value and also if the user sees a logical and monotonic ordering of data throughout time. That is, all the replicas will converge to the same value, eventually. It is not important what happened during the time the client has seen inconsistencies due to several reasons (delay propagation, partition in the system network,...). What we obtain is not to annoy the users with inconsistencies allowing them to continue working using the system and, besides, we have a mask wall to achieve consistent and correct behaviour. In the end, the client finally sees the correct value and perceives a correct behaviour. This new guarantee is considered a consistency model by itself. It is the so called *eventual consistency*. We will talk about it in chapter 5. The application of this concept is broadly used in modern cloud systems. Sacrificing consistent order and timing of data allows to remove several concurrency and version control in the system, making it faster and able to scale at higher rates. This "'trick"' of using apparent consistency with the guarantees is key to modern distributed system, or not so modern since Terry wrote these articles in the earlier 90's and the cloud revolution started more than ten years later. The main explanation for the situation that at the beginning of the first decade of this century is broadly used is the huge impact of the Internet as a domestic facility. Nowadays, almost all machines used by humans can be connected to the Internet. Therefore, the need for techniques that helps to manage with such a huge amount of clients and users of any kind (people, electrical appliances, wearables...) and the study of how inconsistencies can be perceived or repaired without the degradation of availability or application performance is one of the main research areas in current Distributed Systems.

CHAPTER

5 Eventual Consistency

"How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?"

— Sir Arthur Conan Doyle

Eventual consistency has become a famous name last years as it has been used as the main and most known solution to CAP theorem. The recreation of this concept has been evolving and being understood and called with some variations. Firstly used in systems operational oriented, it has been studied to apply it to the transactional environment [Burckhardt et al., 2012]. Practically, there is no complete formal definition for this so called consistency model. There has been some approximations, but they do not cover all cases that the wider definition of it expresses.

In this chapter we argue that eventual consistency is not a consistency model by itself but it is indeed a condition or guarantee that lots of replicated applications will require for a correct performance. If we consider the work of Steinke and Nutt that we have been visiting and commenting along the text since chapter 3, eventual consistency could be considered as another property to define consistency models. It has had an enormous influence in the designing of cloud systems since such systems need to be highly scalable, elastic and tolerable to network partitions.

In the following section we will describe the most remarkable definitions of eventual consistency that we have found through the literature related, even when it was not named as eventual. However, the idea keeps the same condition through time:

> "In the absence of updates, in a quiescent state of the system where all updates have been propagated, all reads will return the same value or all the replicas will share the same state."

Just following those definitions, we will present an study of some formal definitions for it. After that, we explain three representative systems that implement eventual consistency and comment the different approaches and techniques that they have used to construct it. Then, the following section is a collection of the most common mechanisms and some proposed techniques that currently are being developed to ensure that a system is eventual consistent. At this point, when all the basis of eventual consistency is explained, we present a discussion comparing this model with the other consistency models, using the classification and properties defined by Steinke explained in section 3.2. Finally, we summarise all the contents of this chapter and we illustrate the consequences that this consistency model has had in modern -and not so modern- distributed systems.

5.1 Transparency

An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers. A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent. On one hand, replication transparency or that a replicated system act like if is just a single node is a main goal. On the other hand, could be eventual consistency the key to the replication transparency or is indeed the mechanism to achieve it?

5.2 Definitions of Eventual Consistency

First of all, in this section we will write the informal definitions found in the known literature presented here. Almost all of them consider eventual consistency to be a consistency model. Others, say to be a desirable property, as explained just after the definitions. In third place, we present some formal definitions.

5.2.1 Eventual Consistency as a consistency model

The idea of an eventual consistency model appears already in mid seventies, by 1975. It is not named with such a name, but the main concept is captured in Johnson and Thomas [1975]. The RFC recommends to guarantee the copies of the data to be "consistent" with each other. The meaning of this is that given a cessation of update activity to any entry, and enough time for each DBMP (database management process) to communicate with all other DBMPs, then the state of that entry (its existence and value) will be identical in all copies of the database. With this definition we can already glimpse the essence of last definitions of eventual consistency as we will see through this section. The author also warned about the impossibility or difficulties to achieve an identical state all the times in all the replicas given the implicit delay in communications between the copies. Difficulties that will grow with time, number of operations and number of replicas. In this definition, convergence of data is an important feature to keep replication transparency. If users realise that can see a past value, then they may know that there is a replicated server.

The oldest definition has been given in the introduction of this section, just above. For a while, researchers will talk and theorise about eventual consistency without giving it that name. They describe more a system that is wished to be convergent or coherent (do not confuse with Coherent Consistency from 3.1.5).

In Kawell et al. [1988] appears what might be the first time eventual consistency is mentioned in literature. It tells about composing systems and the difficulties of keeping them connected thus affecting data consistency. Describes a system called *Notes*, suggesting to abandon atomic updating, synchronisation and consensus of replicas to allow the system working in case of network partition. Update propagation is performed in background asynchronously. By then, the author already explains that clients may not perceive any inconsistency and, in case it would happen, the clients themselves would notify the incidence to the system administrators. Quoting the exact text:

> *Notes replication ensures* ***eventual consistency*** *of the documents in all replicas: changes made to one copy eventually migrate to all. If all update activity stops, after a period of time all replicas of the database will converge to be logically equivalent: each copy of the database will contain. in a predictable order, the same documents; replicas of each document will contain the same fields. In other words, a program cannot detect the difference between replicas through the document manager programmatic interface.*

In a project named *Bayou*, Terry et al. [1994] say that weak consistency, considered as the one that allows the data replicas to vary and be accessed in the meanwhile, is implemented. Besides, it also desires for Bayou, and practical systems, for the replicas to be convergent when all updates are propagated and no more changes are generated. Literally: "Practical systems generally desire *eventual consistency* in which servers converge towards identical database copies in the absence of updates. It relies on two properties: total propagation and consistent ordering."

Saito and Shapiro [2005] informally say that eventual consistency guarantees only that the state of replicas will eventually converge, allowing inconsistencies to be observed by applications or users in optimistic replicated systems, where availability is the most important goal. They consider it to be the weakest level of consistency considering the degree of replica blocking accesses when some restrictions are not met, named *bounded divergence*. In the other segment, the stronger consistency, will be single-copy. To implement an optimistic replicated system that ensures eventual consistency, Saito proposes that it will be necessary to compute operation ordering while identifying and solving conflicts and committing operations.

About Vogels definition [Vogels, 2009], considered a form of weak consistency, it is remarkable two aspects of it. The first, his definition reminds to a a guarantee for the users, similar to the ones described by Terry et al. [1994]. Quoting: "the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value.". This definition focuses more in what the users perceive than what is really happening in the system. As an example, let us imagine a situation where a network partition has taken place. To simplify, we are going to consider only two partitions of the system A and B, but it could be more than just two. There is a client who accesses repeatedly to the same data in partition A. The very same data is being modified meanwhile in partition B by other client once and then it stops. The client in partition A will read all the time the same value for that precise data, which is different than the value in the other partition and there are no more updates to that data. Hence, the system has not reached convergence, but the client in partition A is reading all the time the same value like the definition of eventual consistency given by the author. After that, when the partition heals, the data will be updated and converged. The second interesting aspect is the definition of what the author calls *inconsistency window*, and defined as how long would the data be stale until it gets its last or correct value. For the good behaviour of the system it is very important to calculate or speculate the maximum size of the inconsistency window, considering that no failures occur during the process, using factors like communication delays or latency, system load or the number of nodes or replicas involved in the system.

However, Vogels' definition does not meet the requirement for the definition enunciated by Terry et al. [1994] and others. It certainly remains to the consistency perceived by the user explained in chapter 4, because it focuses more in the returned value to the client instead of the state of the system. Certainly, we could differentiate this definition from the others and highlight it as *client centric eventual consistency*. It is not so important that data finally have the same value, that replicas converge, but to make the users think that they are working with correct data. One could ask what it happens if due to this new kind of inconsistency, something goes wrong and the resulting service of the client fails. Vogels proposes to add what is called the *human factor*, in other words, to allow the users and the representative agent of the service that the system provides to communicate and solve the situation. The perfect image or behaviour of the system is not so important and users are supposed to tolerate this kind of situations. Systems may have become more modern, so the society too. People are more used to work with computers and devices on the internet constantly. Therefore, it is not so bad idea to let the users to inform to the administrators about a misbehaviour of the system. This is another fundamental change on the way systems evolve, implying more the clients in the process.

In Fekete and Ramamritham [2010] eventual consistency is defined as a consistency model that is tolerant to network system partitions allowing the system to be highly available. The system can work with what the authors call "disconnected operations". That is, once the client has access to one node, even in the environment of a partition, the client is able to continue working with the connected server. Once the possible partition is healed, the node only has to send the updates using the gossip protocol. If conflicts appear, then mechanisms implemented to solve them will be applied. The authors define *eventual consistency* explaining the main implementation techniques, i.e., to allow to any client to access to any replica reachable (update anywhere, anytime) and then propagate the updates in the background using gossip communication protocols. Considering that not a lot of operation are commutative or idempotent and no sequencer seems to be implemented, a reordering of the operations applied to the replicas will be necessary to solve conflicts. This can imply to undo the effect of certain conflicted operations and even some operation could have not been applied. A control of the time that the replica values may differ is also commented as bounded divergence, which reminds to the *inconsistency window* defined by Vogels. Once the order of the operation is fixed, the name of the sequence of the correct operation or prefix is called "agreed past". Thus, all replicas finally have agreed pasts to all the executions that are equivalent to a sequential order of operations. Besides, it talks about "session properties" to complement the mechanisms mention before. Those properties are the session guarantees explained in chapter 4. The authors also consider causal consistency (see section 3.1.3) to be a restrictive implementation of eventual consistency since it adds the causal relation among operations to partially order operations rather than consider it a different consistency model. We can start to suspect that eventual consistency is like some kind of complement or characteristic of certain consistency models and not a model instead.

Similar to Saito's definition, Shapiro and Kemme [2009] defines what we could call *strong eventual consistency*. Informally, it requires that all replicas of an object will eventually reach the same, correct, final value, assuming that no new updates are submitted to the object. Like a standard definition of a consistency model, they define it as a sequence partially ordered of operation schedule for each process that belongs to the system. The particularity of these sequences is that they have the property of being equivalent and not necessarily sequential, denoting the use of commutative operations that do not affect to the final result. Thus, all the data is convergent. More exactly, assuming that all replicas have the same initial state, they define eventual consistency of a replicated object with four conditions that have not to be violated in any situation:

- There is a prefix of the schedule S_n^x that is state-equivalent to a prefix of the schedule S_n^x, of any other node n' holding a replica of x. Such a prefix is called a *committed prefix* of S_n^x.
- The committed prefix of S_n^x grows monotonically through time, i.e., the set of operations and their relative order remains unchanged.
- For every operation w_i submitted by a user, either w_i or $\overline{w_i}$ eventually appears in the committed prefix of S_n^x (but not both, and not more than once)
- An operation w_i in the committed prefix satisfies all its preconditions (e.g., the state of the object immediately before the execution of the operation fulfils certain conditions).

Last definition we will consider in this section is the one made by Bailis and Ghodsi [2013]. If no additional updates are made to a given data item, all reads to that item will eventually return the same value. Of course, until that moment of convergence, read operations may return non correct values and the system will be still eventually consistent. The moment of no more updates are done will arrive at some point in the future thus, the system still is eventually consistent. Authors defend that eventual consistency is more about a liveness property, that is, “something good eventually happens”. However, eventual consistency definition does not say anything about safety property meaning “nothing bad happens”. This part of the equation is the responsibility of the consistency model chosen for the system and the access restrictions that implements if data is not correct.

If we consider this last definition, the convergence property explained in Gray et al. [1996], the technical report of Mahajan et al. [2011], work that considers convergence as an aspect differentiated from consistency and even Fekete and Ramamritham [2010] says openly that we are treating with a liveness property instead of a consistency model. In addition to those works, we add our envision that eventual consistency is a property that some consistency models will accomplish (see section 3.4) if the rules agreed with the data store convergence is included. To understand it better, we have used Steinke and Nutt unified theory of consistency models and inferred that those models that have at least GDO as a guarantee granted, the system will be naturally convergent.

5.2.2 Eventual Consistency and a formal definition

Almost all the formals definitions that we have found are based in the definition of a *prefix* of operations that all replicas have to agree in order to converge. Following directly this approach we have papers like Saito and Shapiro [2005], Shapiro and Kemme [2009] among others. Fekete and Ramamritham [2010] calls it "agreed past". The problem with this style of definition is that they leave out situations that the replica may not share the same prefix but they will have the same value by using a reconciliation technique that does not modifies the prefix. Even Bosneag and Brockmeyer [2002], that presents a formal definition using an algorithmic approach, still considers history of operations taking as the principal history the master's replica, which is similar to the concept of prefix. In Bouajjani et al. [2014] a formal verification of eventual consistency is presented, but only in optimistic replicated environments. As we seen in the following section, optimistic replication (see section 2.4.1) is one of the most used techniques to implement a highly scalable system.

As far as we have presented here, no formal model seems not to have enough expressivity to formally define eventual consistency. If the rules presented in the consistency model imply data convergence, the definition of eventual consistency is implicit in those rules. If it does not imply data convergence, we will need to force with extra techniques and reconciliation mechanisms that will help with data convergence as well as recovery from partitions and other failures that come with data or updates loss.

5.3 Implementation and techniques

In this section we will present the most common tools, mechanisms, implementation, replication and reconciliation techniques implemented to achieve data convergence or eventual consistency. The majority of the mechanisms that we talk about are taken mainly from Bernstein and Das [2013], Kawell et al. [1988], Terry et al. [1994], Vogels [2009] and F.D. Muñoz-Escoí and Esparza-Peidro [2011].

First of all, mention that the most replication algorithm used is optimistic replication. Broadly explained in **??**, its basic mechanism is to spread the update to the other nodes or replicas of the system at the same time that the interaction with the client is taken place, instead of confirming that all replicas have been updated before answering the client. It is a particular type of passive replication model, but with the approach of update anywhere anytime. Any server is accessed by a client, not necessarily the master, and performs the tasks that the client requires. In the meanwhile, updates are propagated using background gossip like communication protocols. Of course, choosing only one node to work with can be dangerous for durability and some implementations uses at least a minimum number of nodes to perform the update, named quorum.

As we have shown in chapter 3, section 3.4, the consistency models that require all these mechanisms are those that are not naturally convergent. As we checked in the lattice 3.9, causal and PRAM models are those considered fast and also not convergent. The most used model is the causal one. To solve conflict situations, there are several ways, like *Thomas write rule*, to use commutativity when possible, to study mergeable situations (development of CRDTs [Shapiro et al., 2011]).

The system that has been very famous that implements a subset of these techniques is Dynamo [DeCandia et al., 2007]. Inspired in a P2P system called Chord ([Stoica et al., 2003]), it implements a highly scalable system that uses an approach of causal consistency by means of versioning and conflict resolution to ensure eventual consistency.

5.4 Conclusions

As show along the chapter, eventual consistency has been tried to be defined as a consistency model but, as we can see, since defined in Johnson and Thomas [1975], it is is a characteristic desirable for a replicated system. But it can vary depending on the necessities or requirements of the system. however it is indeed a liveness condition desirable if we want convergent data. So, as shown with Steinke properties, if a data centric consistency model meets GDO or GAO properties, it will converge by itself. Otherwise, extra mechanisms have to be added in order to achieve convergence, if desired.

Classical literature explains that strong consistency usually implies strong synchronisation, like global synchronisation. Thus, it is obvious that to scale a strong consistent system will be complicated and will have a heavy behaviour and long response time since it requires the synchronisation of all the nodes of the system. That is the main reason that techniques to achieve the liveness property of convergence or eventual consistency typically leave the synchronisation part of the updating replica process for later, for another moment. Making not mandatory to update everywhere as fast as possible but just when is needed and, if not yet done, to give the impression that is synchronise or to develop a service that does not require these kind of behaviour and can work with a more relaxed one. "the principle is we will fix the conflict, the information will arrive, but later, when it is really necessary, to execute operations that depends on the result or to give a response to the clients".

Eventual consistency is, informally speaking, to allow our system to be inconsistent for some period of time giving the guarantee that will arrive to a unique image data and not worrying if some inconsistencies are perceived during this time or inconsistency window because they will be treated or masked by some techniques, such as client centric consistency guarantees and trusting the reconciliation techniques that have been implemented for it. We could say that we do not take the advise of "prevention rather than cure", which be a pessimistic vision of the behaviour of the system, and do "instead of preventing, treating". This is the price we pay to obtain wide area distributed systems and a solution to manage CAP Theorem.

Chapter

6 Conclusions

"It is the time you have wasted for your rose that makes your rose so important."
— Antoine de Saint-Exupéry, *The Little Prince*

We have presented the problematic showed with the CAP theorem in chapter 1 and propose eventual consistency as a particular solution to solve or to live with this situation in case of partitions. After some common knowledge about distributed systems in the chapter 2, we have defined the data oriented consistency models and given a formalism that helps to compare consistency models among them. From this chapter, we have inferred that the consistency mode chosen can affect to the system in several manners, being the most remarkable:

- The programming style, more synchronous or asynchronous.
- Partition tolerance.
- Reconciliation techniques and conflict resolution.
- Data convergence, also understood as if the system will be eventually consistent.
- System scalability.
- Availability.
- The system latency.

Therefore, it is extremely important to chose wisely the kind of consistency model we want in our distributed application since it will affect in all previous characteristics and others that we could have leave out of the list.

From the reasoning about how synchronisation restricts systems behaviour and the time that a value is allowed to be stale, which implies to have synchronisation barriers, we can envision that what makes a consistency model, client or data oriented, really strong is in function of how constrained these time gaps or periods are. The more strict the operations of updating are, the more synchronised the system is. Just with having a look to the strongest consistency model, that is the strict or atomic model, that obliges to the operation to be updated automatically, without any latency. So the key to let scalability to arise was to delay the synchronisation moment of the correct value of replicated data to the last moment, or to the really needed moment (it is really necessary to have all values of all replicas updated if they are not going to be accessed or used?). This situation directly affects to the data consistency but it does not say anything about if data will be eventually consistent or coherent. We have presented here Steinke and Nutt formalism to define consistency models and conclude that only the models that respect GAO or GDO properties will be automatically eventually consistent. For the other models that do not follow one or the other property, will not be naturally convergent and extra healing mechanisms will need to be added to the systems. Unfortunately, the models remaining are those called fast consistency models, that is, models that can each node belonging to the system can make decisions about operations without needing extra information or collaboration, which implies normally synchronisation

We have defended that to have a scalable system, some times stale data may be perceived by the users while mechanisms to convergence are working. A way to mask the behaviour we want the users to perceive are the session guarantees and the concept of session presented in chapter 4. However, it is not so terrible that sometimes users see this malfunctioning of the system if they are said that such behaviour is not perceive as wrong.

Finally, in chapter 5, some of the most known definitions for eventual consistency have been presented, ordered in time. We conclude that eventual consistency is not in fact a consistency model by itself, like the ones presented in chapter 3. It does not follow the same rules to be defined and we argued that it is a liveness property of the system. From it we interpret that, if we want a highly scalable distributed system to perform properly, designers have to ensure that the property of convergence or eventual consistency is finally achieved. Having this in mind, eventual consistency can be understood as a set of techniques to solve or live with the CAP Theorem situation.

Future work

From this work, two interesting lines are suspect to be researched in a future work. The first, to research on looking for a formal definition of eventual consistency using some sort of temporal logic or a kind of algebra that is expressive enough to show time in the definition. Secondly, other interesting point to investigate is to proof formally what is the basic consistency model that arises from each communication protocol without any extra mechanisms.

Bibliography

Ahamad, M., Bazzi, R. A., John, R., Kohli, P., and Neiger, G. (1993). The power of processor consistency. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 251–260, New York, NY, USA. ACM.

Alvaro, P., Bailis, P., Conway, N., and Hellerstein, J. M. (2013). Consistency without borders. In *SoCC*, page 23.

Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., and Ciarfella, P. (1995). The totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4):311–342.

Attiya, H. and Friedman, R. (1996). Limitations of fast consistency conditions for distributed shared memories. *Inf. Process. Lett.*, 57(5):243–248.

Bailis, P. and Ghodsi, A. (2013). Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63.

Bernstein, P. A. and Das, S. (2013). Rethinking eventual consistency. In *SIGMOD Conference*, pages 923–928.

Birman, K. P. (1986). ISIS: a system for fault-tolerant distributed computing. Technical report, Cornell University, Department of Computer Science.

Bosneag, A.-M. and Brockmeyer, M. (2002). A formal model for eventual consistency semantics. In *IASTED PDCS*, pages 204–209.

Bouajjani, A., Enea, C., and Hamza, J. (2014). Verifying eventual consistency of optimistic replication systems. In *POPL*, pages 285–296.

Brewer, E. A. (2000). Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA. ACM.

Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. (1992). Distributed algorithms, 6th international workshop, wdag '92, haifa, israel, november 2-4, 1992, proceedings. In *WDAG*, pages 362–378.

Burckhardt, S., Leijen, D., Fähndrich, M., and Sagiv, M. (2012). Eventually consistent transactions. In *ESOP*, pages 67–86.

Cantin, J. F., Lipasti, M. H., and Smith, J. E. (2005). The complexity of verifying memory coherence and consistency. *IEEE Trans. Parallel Distrib. Syst.*, 16(7):663–671.

Chockler, G., Keidar, I., and Vitenberg, R. (2001). Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469.

Cholvi, V. and Bernabéu, J. (2004). Relationships between memory models. *Inf. Process. Lett.*, 90(2):53–58.

Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. (2008). Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288.

Davidson, S. B., Garcia-Molina, H., and Skeen, D. (1985). Consistency in partitioned networks. *ACM Comput. Surv.*, 17(3):341–370.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: amazon's highly available key-value store. In Bressoud, T. C. and Kaashoek, M. F., editors, *SOSP*, pages 205–220. ACM.

Demers, A. J., Greene, D. H., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H. E., Swinehart, D. C., and Terry, D. B. (1987). Epidemic algorithms for replicated database maintenance. In *PODC*, pages 1–12.

Défago, X., Schiper, A., and Sergent, N. (1998). Semi-passive replication. In *SRDS*, pages 43–50.

F.D. Muñoz-Escoí, J.R. García-Escrivá, M. P.-L. and Esparza-Peidro, J. (2011). Managing Scalable Persistent Data. Technical report, Institut Universitari Mixt Tecològic d'Informàtica, Universitat Politècnica de València.

Fekete, A. D. and Ramamritham, K. (2010). Consistency models for replicated data. In *Replication*, pages 1–17.

Fischer, M. J., Lynch, N. A., and Paterson, M. (1985). Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P. B., Gupta, A., and Hennessy, J. L. (1990). Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA*, pages 15–26.

Gilbert, S. and Lynch, N. A. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59.

Goodman, J. R. (1989). Cache consistency and sequential consistency. Technical report, IEEE Scalable Coherent Interface Working Group.

Gray, J., Helland, P., O'Neil, P. E., and Shasha, D. (1996). The dangers of replication and a solution. In Jagadish, H. V. and Mumick, I. S., editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 173–182. ACM Press.

Herlihy, M. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492.

Homburg, P., van Steen, M., and Tanenbaum, A. S. (1996). An architecture for a wide area distributed system. In *ACM SIGOPS European Workshop*, pages 75–82.

Hutto, P. W. and Ahamad, M. (1990). Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In *ICDCS*, pages 302–309.

Johnson, P. and Thomas, R. (1975). The maintenance of duplicate databases. RFC 677.

Kawell, Jr., L., Beckhardt, S., Halvorsen, T., Ozzie, R., and Greif, I. (1988). Replicated document management in a group communication system. In *Proceedings of the 1988 ACM Conference on Computer-supported Cooperative Work*, CSCW '88, pages 395–. ACM.

Kopetz, H. and Veríssimo, P. (1993). *Distributed Systems*, chapter Real Time and Dependability Concepts, pages 411–444. Volume 1 of Mullender [1993], second edition edition.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691.

Lamport, L. (1986a). The mutual exclusion problem: partii - statement and solutions. *J. ACM*, 33(2):327–348.

Lamport, L. (1986b). On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–85.

Mahajan, P., Alvisi, L., and Dahlin, M. (2011). Consistency, availability, convergence. Technical Report TR-11-22, Computer Science Department, University of Texas at Austin.

Mosberger, D. (1993). Memory consistency models. *Operating Systems Review*, 27(1):18–26.

Mullender, S., editor (1993). *Distributed Systems (2Nd Ed.)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Oppen, D. C. and Dalal, Y. K. (1983). The clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Trans. Inf. Syst.*, 1(3).

Powell, D. (1993). Distributed fault tolerance - lessons learned from delta-4. In *Hardware and Software Architectures for Fault Tolerance*, pages 199–217.

R.J. Lipton, J. S. (1988). PRAM: A scalable shared memory. Technical report, Princeton University.

Saito, Y. and Shapiro, M. (2005). Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319.

Schneider, F. B. (1993). *Distributed Systems*, chapter What Good Are Models and What Models Are Good, pages 17–26. Volume 1 of Mullender [1993], second edition edition.

Shapiro, M. and Kemme, B. (2009). Eventual consistency. In Liu, L. and Özsu, M. T., editors, *Encyclopedia of Database Systems*, pages 1071–1072. Springer US.

Shapiro, M., Preguiça, N. M., Baquero, C., and Zawirski, M. (2011). Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88.

Steinke, R. C. and Nutt, G. J. (2004). A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849.

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. (2003). Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32.

Tanenbaum, A. S. and van Steen, M. (2007). *Distributed systems - principles and paradigms (2. ed.)*. Pearson Education.

Terry, D. B., Demers, A. J., Petersen, K., Spreitzer, M., Theimer, M., and Welch, B. B. (1994). Session guarantees for weakly consistent replicated data. In *PDIS*, pages 140–149.

Vogels, W. (2009). Eventually consistent. *Commun. ACM*, 52(1):40–44.