Nuno Filipe Estêvão Gomes

Bachelor in Computer Science and Engineering

A Semantic Consistency Model to Reduce Coordination in Replicated Systems

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

Master of Science in
Computer Science and Engineering

Adviser:	Carla Ferreira, Associate Professor, NOVA University of Lisbon

Examination Committee

Chairperson:	Prof. Miguel Goulão, FCT/UNL
Raporteur:	Prof. João Barreto, IST/UL
Member:	Prof.ª Carla Ferreira, FCT/UNL

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2020

A Semantic Consistency Model to Reduce Coordination in Replicated Systems

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my family. Thank you all.

Acknowledgements

I would like to start by thanking my advisor Prof. Carla for giving me the opportunity to work on this dissertation, to guide me since day one, and always be available, not only in the conception and development of the work but also for the help provided in the very important and exhaustive reviewing of the writing of the dissertation. I also want to thank my co-adviser André for his help in the elaboration phase of this dissertation, in which the help provided was crucial to reach what we achieved. This work was partially supported by the project PTDC/CCI-INF/32081/2017 - 2, to which I would like to thank for the grants given.

Additionally, I would like to thank all the FCT teachers I have had over the five years, as well as all the colleagues who have always accompanied me, who were so important so that I could come this far.

Finally, I dedicate this work to my family, in special to my parents and brother, who have given me the opportunity to study away from home, who have always been there, and for everything they have done for me all over the years. You are the best.

Abstract

Large-scale distributed applications need to be available and responsive to satisfy millions of users, which can be achieved by having data geo-replicated in multiple replicas.

However, a partitioned system cannot sustain availability and consistency at fully. The usage of weak consistency models might lead to data integrity violations, triggered by problematic concurrent updates, such as selling twice the last ticket on a flight company service. To overcome possible conflicts, programmers might opt to apply strong consistency, which guarantees a total order between operations, while preserving data integrity. Nevertheless, the illusion of being a non-replicated system affects its availability. In contrast, weaker notions might be used, such as eventual consistency, that boosts responsiveness, as operations are executed directly at the source replica and their effects are propagated to remote replicas in the background. However, this approach might put data integrity at risk. Current protocols that preserve invariants rely on, at least, causal consistency, a consistency model that maintains causal dependencies between operations.

In this dissertation, we propose a protocol that includes a semantic consistency model. This consistency model stands between eventual consistency and causal consistency. We guarantee better performance comparing with causal consistency, and ensure data integrity. Through semantic analysis, relying on the static analysis tool CISE3, we manage to limit the maximum number of dependencies that each operation will have. To support the protocol, we developed a communication algorithm in a cluster. Additionally, we present an architecture that uses Akka, an actor-based middleware in which actors communicate by exchanging messages. This architecture adopts the publish/subscribe pattern and includes data persistence. We also consider the stability of operations, as well as a dynamic cluster environment, ensuring the convergence of the replicated state. Finally, we perform an experimental evaluation regarding the performance of the algorithm using standard case studies. The evaluation confirms that by relying on semantic analysis, the system requires less coordination between the replicas than causal consistency, ensuring data integrity.

Keywords: distributed systems, replication, synchronization, consistency, concurrency, invariants, semantic analysis, causality

Resumo

Aplicações distribuídas em larga escala necessitam de estar disponíveis e de serem responsivas para satisfazer milhões de utilizadores, o que pode ser alcançado através da geo-replicação dos dados em múltiplas réplicas.

No entanto, um sistema particionado não consegue garantir disponibilidade e consistência na sua totalidade. O uso de modelos de consistência fraca pode levar a violações da integridade dos dados, originadas por escritas concorrentes problemáticas. Para superar possíveis conflitos, os programadores podem optar por aplicar modelos de consistência forte, originando uma ordem total das operações, assegurando a integridade dos dados. Em contrapartida, podem ser utilizadas noções mais fracas, como a consistência eventual, que aumenta a capacidade de resposta, uma vez que as operações são executadas diretamente na réplica de origem e os seus efeitos são propagados para réplicas remotas. No entanto, esta abordagem pode colocar em risco a integridade dos dados. Os protocolos existentes que preservam as invariantes dependem, pelo menos, da consistência causal, um modelo de consistência que mantém as dependências causais entre operações.

Nesta dissertação propomos um protocolo que inclui um modelo de consistência semântica. Este modelo situa-se entre a consistência eventual e a consistência causal. Garantimos um melhor desempenho em comparação com a consistência causal, e asseguramos a integridade dos dados. Através de uma análise semântica, obtida através da ferramenta de análise estática CISE3, conseguimos limitar o número de dependências de cada operação. Para suportar o protocolo, desenvolvemos um algoritmo de comunicação entre um aglomerado de réplicas. Adicionalmente, apresentamos uma arquitetura que utiliza Akka, um *middleware* baseado em atores que trocam mensagens entre si. Esta arquitetura utiliza o padrão *publish/subscribe* e inclui a persistência dos dados. Consideramos também a estabilidade das operações, bem como um ambiente dinâmico de réplicas, assegurando a convergência do estado. Por último, apresentamos a avaliação do desempenho do algoritmo desenvolvido, que confirma que a análise semântica das operações requer menos coordenação entre as réplicas que a consistência causal.

Palavras-chave: sistemas distribuídos, replicação, sincronização, consistência, concorrência, invariantes, análise semântica, causalidade

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This initial chapter introduces the context of the dissertation and discusses the motivation behind it. Also, we present the proposed solution and outline the skeleton of the remaining document.

1.1 Context

Most of the applications or services currently in use, provided on the Internet, have the purpose of reaching thousands or millions of users. These applications are expected to be always functioning properly, i.e., without failures and highly available, not having delays, so they can offer the best user experience in terms of performance. This idea is feasible if they are distributed, relying on geo-replicated storage systems, containing numerous replicas scattered across the globe, considered to be replicated and fault-tolerant [7, 8]. However, high availability comes with a trade-off: consistency. As stated in the CAP theorem [19] (detailed in Chapter 2), a partitioned system cannot fully sustain both availability and consistency. Instead, it is necessary to choose the most appropriate one regarding the application needs.

To preserve data integrity in all replicas, synchronization is required between them, increasing transaction's communication latency and, consequently, decreasing performance. Programmers can follow several approaches when it comes to consistency, such as strong or weak consistency models.

On one hand, opting for weaker assumptions guarantees that updates are propagated asynchronously, executed directly at the source replica, and then propagated in the background, ensuring availability, i.e., boost responsiveness [6, 15, 51, 52]. Nonetheless, application correctness might become vulnerable since replicas' state may diverge, and invariants might not be preserved due to the execution of concurrent operations that can lead to data integrity violations. These violations occur if concurrent operations conflict, and/or have dependencies, i.e., operations that do not commute are propagated and executed in a different order. For instance, consider a bank application, where the primary operations are deposits and withdrawals, and the invariant states that the balance should always be non-negative. If both Alice and Bob share the same bank account, whose balance is 100 euros, and both perform concurrent withdrawals of 60 euros, it will result in a negative balance, thus breaking the invariant.

In contrast, strong consistency guarantees convergence and data integrity because it requires a total order of operations in all replicas [5, 6, 8]. Although this allows them to converge to the same state, the system's availability is affected. The reason is that it requires frequent synchronization. Additionally, instead of applying a single model to the whole application, the architect of the system may choose to apply weak or strong models for each operation [6, 8, 27]. Those that, in concurrency, do not conflict with other operations, can be executed immediately, while others require synchronization.

1.2 Motivation

A replicated system cannot sustain both availability and consistency, as it was mentioned earlier, so some systems rely on weak consistency models to increase availability. However, the correctness of these systems can be affected. In contrast, previous works have proposed applying different consistency models to different operations. Even so, programmers have to reason on the effects of each operation to determine those that need synchronization, which is difficult.

Current protocols for weakly consistent geo-replicated databases that ensure global invariants, which are conditions that should be true throughout the program life cycle, either rely on causality or have a global synchronization point. The main challenge is to find a balance between availability and consistency, where programmers do not have to worry about problematic concurrent updates, ensuring data integrity preservation, as well as defining a non-blocking synchronization point among replicas.

1.3 Proposed Solution

In this dissertation, we aim to propose a synchronization protocol that reduces synchronization effort and relieves the programmer from reasoning about possible problems related to operations that are executed concurrently over a geo-replicated database.

The protocol relies on CISE3 [42], a static analysis tool. Based on an application specification, the tool identifies which operations are problematic when executed concurrently, that is, may not preserve the integrity invariant. The tool also provides the causal relationship between operations. If operation A is causally dependent on operation B, then the execution of B must precede the execution of A. Otherwise, integrity invariant may also be broken. The protocol will receive as input a set of conflicting operations and a set of causal dependencies between them. In many solutions presented in the literature, the programmer, besides reasoning about conflicting pairs of operations, also needs to define a synchronization protocol that avoids or somehow prevents conflicts from happening. In the bank application example, where clients can make deposits to an account and withdrawals from, the integrity invariant states that the balance must be non-negative. The `withdraw` operation conflicts with other withdrawals since the invariant might not be maintained (e.g., both Alice and Bob share the same account and both withdraw the total balance of that account, leaving it negative). A programmer may choose to disallow two withdrawals from executing concurrently. However, it is not the best approach, as withdrawals from different bank accounts are not problematic. In this scenario, the bank account could be one of the parameters of the withdrawal operation.

In this sense, we intend to perform a semantic analysis of the operations, which includes parameter analysis, proposing a consistency model called **Semantic Consistency**. Through this analysis, we can reduce and limit the maximum size of causal dependencies that a specific operation has throughout the execution of the system. In case an operation `A` has x dependencies, then the replica `r1` that intends to execute `A` can only proceed with the execution if it has previously executed the x dependencies. This notion is weaker than the one provided by causal consistency. The reason is that, upon receiving the operation `A` from a replica `r2`, it requires the execution of all the updates that occurred in `r2` before `A`, to apply `A`. Causal consistency defines a partial order between events or operations. We intend to reduce this partial order while ensuring system correctness.

We seek to define a protocol in an abstract way, which includes the architecture to support the implementation that constitutes the semantic consistency model. Also, we want to formally prove that the abstract solution works, ensuring data integrity preservation in applications built on a geo-replicated setting, reducing coordination among replicas. Additionally, the development of the concrete solution, developing an algorithm and its implementation, and later its validation with certain case studies.

1.4 Document Structure

The rest of the document is organized as follows:

- **Chapter 2 - Background:** Provides useful concepts as a basis for understanding the following chapters, with an emphasis on consistency models, replication, application correctness, and causality tracking in a distributed system.

- **Chapter 3 - Related Work:** Presents related work on handling concurrent conflicts and dependencies between operations. Moreover, features a join model regarding the integration of new replicas in an asynchronous distributed system. Finally, provides a brief conclusion of the chapter.

- **Chapter 4 - Design and Implementation:** This chapter gives an overview of the implementation and the architecture that supports it. Also, introduces the construction of the graph of dependencies using a semantic analysis provided by the static analysis tool CISE3 [42].

- **Chapter 5 - Dynamic Cluster:** Upon an understanding of the architecture and the implementation, this chapter covers how we implemented a dynamic environment where new replicas join the cluster. It also explains the stabilization of operations and how stability is handled in this scenario.

- **Chapter 6 - Evaluation:** Features an experimental evaluation of the performance of the algorithm, comparing execution times, according to the number of operations performed, latency, and consistency model, for two case studies.

- **Chapter 7 - Conclusion:** Finally, we discuss and summarize what was accomplished in this dissertation and the conclusions we derive from our approach. Additionally, we also list the contributions of the dissertation and discuss future work.

Chapter 2

Background

In this chapter key concepts are presented for a better understanding of some notions used throughout this thesis. Starting with main transaction's properties, why replication is important, the famous CAP theorem, following by a summary explanation about existing consistency models, what are CRDTs and, lastly, events and properties that happen or exist while executing operations concurrently.

2.1 ACID Properties in Transactions

A transaction is a set of instructions to be processed in a database management system, that can manipulate data of a database. For example, it may change its contents in case of writing operations or leaving it unmodified if it only reads. A transaction succeeds if all its operations also succeed. Considering a single database, to maintain its consistency every transaction that interacts with the latter must ensure data integrity. This can be achieved if a transaction presents ACID properties, which stands for Atomicity, Consistency, Isolation and Durability [41, 44]:

- **Atomicity** states that a transaction will commit successfully iff all single operations succeed: "all or nothing rule".
- **Consistency** refers to the state of the database, meaning that before and after executing a transaction, the database must preserve data integrity constraints.
- **Isolation** means that a transaction acts as being the only one executing, not having the perception that other transactions might be executing concurrently.
- **Durability** consists in the preservation of the effects of a successfully committed transaction, even if any failures occur afterward.

In terms of implementation, concurrency control protocols are implemented to ensure Atomicity and Isolation, using locking mechanisms [44]. The two-phase commit protocol (2PC) is an example of a concurrency control protocol that it's used in distributed databases to coordinate all participants on whether all can commit the transaction or not [41]. Durability can be ensured using database backups, for instance.

2.2 Replication

When thinking about a distributed system we can imagine that it has multiple processes in a distributed way, i.e., in different machines or, as we should say, replicas or nodes. Each process is responsible for managing operations, and failures can arise. For this reason, to ensure the system operates as expected despite possible failures, replication is used. Communication between clients and replicas is accomplished exchanging messages [55].

The choice of how to replicate is influenced by several factors, such as the frequency of the usage of the system, fault model involved, etc. There are several replication models that can be used, two of them are synchronous and asynchronous models. While the former is adopted in systems that provide strong consistency, the latter is used in weaker consistency systems [55] (see Section 2.4 about consistency models).

In the **synchronous model**, an operation cannot execute immediately in a given replica (source). Firstly, needs to synchronize with the others, then executes it. Well, this can delay the performance due to higher latency that the client will experience. Total order broadcast [56] technique is used to maintain a total order of all operations that are propagated between replicas.

In contrast, from a client perspective, the **asynchronous model** presents as being faster than the synchronous model, due to the immediate execution on the replica it was requested. The result is presented to the client, while the propagation to other replicas happens in background. Consequently, it can lead to inconsistent states, when read operations present different results (see Section 2.4), and replicas take more time to converge, compared with the synchronous model.

There are also two other strategies for replication: operations-based and state-based replication [51]. Using **operations-based (or active) replication**, the state in all replicas is continuously updated, so the probability of losing updates is low. The source replica (where the request was made) propagates the operations' effects to other replicas, which will then execute them so that all can share the same state. On the other hand, working with **state-based (or passive) replication**, the state of the source replica is propagated rather than operations. Then receivers have to merge their state with the one received, to prevent loss of operations' effects.

2.3 CAP Theorem

The CAP (Consistency, Availability, Partition-tolerance) theorem [19] refers that only two of the three following characteristics can be fully ensured in a distributed system:

- **Strong consistency:** All clients observe the same data at a specific point of time. This is achievable if a total order on all operations is defined in all replicas. Whenever a write operation succeeds in one replica, it must be replicated in the others right away. This type of consistency will be explained in detail in Section 2.4.
- **Availability:** Reaching availability requires the system to remain always up. While making a request, a response is eventually received, even if one or more replicas are not responding.
- **Partition-tolerance:** The network is divided in different partitions. This property states that the system continues executing operations despite failures that might take place between any partitions.

Consequently, a distributed system can be categorized as follows: CP (Consistent and Partition-tolerant), CA (Consistent and Available) or AP (Available and Partition-tolerant). Having partitions is crucial to improve responsiveness in a geo-replicated system [7]. Thus, Partition-tolerance is unavoidable, which means most systems are either CP or AP.

2.4 Consistency Models

For a programmer, reasoning about which type of consistency to choose from is not straightforward. While one can think "*Let's use strong consistency and ignore all concurrency problems*", other can state "*We can use weak consistency and hope that everything goes as expected*". These are two different points of view, covering both extremes when it comes to consistency models.

Strong consistency is the most strict type of consistency. Nevertheless, it makes easy to reason about the evolution of the system, since all replicas will have the same state, as if only one replica existed. However, replicas will require more time to coordinate. Warranties [38], homeostasis protocol [49] and predictive treaties [39], focus on applications that are built on top of strongly consistent systems, suggesting a way to increase performance (discussed in Chapter 3).

Linearizability and Serializability are two of the models that serve strong consistency. **Linearizability** reasons about single operations (e.g., reads or writes). Writes should appear to take effect right away at some point in time between its invocation and its response [2]. **Serializability** refers to the Isolation property in ACID (see Section 2.1). Serializability guarantees that a concurrent execution of multiple transactions, commonly containing read and write operations, is equivalent to some serial execution of the transactions, i.e., as if each transaction executed in sequential order, leading to the same final result. A serializable execution of a set of transactions will preserve application correctness if every single operation also preserves it [2].

Not all operations require strong consistency to achieve system's correctness, concretely commutative operations, which can be reordered between different nodes, increasing performance (e.g., two operations, A and B, executing concurrently; A depends on B, which means they don't commute either, so effects of B must be executed before A). This is where weak consistency models enter.

Eventual consistency is the weakest consistency model. It guarantees availability without a total order of operations between replicas. When a replica receives a request to execute a particular operation, it first executes it, then presents the result to the client and finally, propagates the effect to all other replicas. If clients stop making write requests, all replicas will eventually receive all operations' effects and converge to the same state. As previously stated, the fact that eventual consistency does not maintain a total order between operations can be a problem, due to the lack of commutativity between concurrent operations, which may lead to inconsistencies [7, 15]. The eventual consistency model became popular due to Dynamo [16], a highly available key-value storage system developed by Amazon, which despite having multiple replicas, operations are executed by contacting only a few of them.

Another weak model is **causal consistency**. Unlike eventual consistency, there are no guarantees that, if no more write operations appear, all replicas will converge to the same state. Nevertheless, ensures that reads will follow causality between writes: "if one event "influences" a subsequent operation, a client cannot observe the second without the first" [4]. It ensures causal order of operations, but not total order, which may lead to integrity violation. Previous work, such as RedBlue consistency [27], and Indigo [6], a middleware library that guarantees the preservation of invariants by reasoning in which operations that require coordination and using locking mechanisms to avoid conflicts, requires causality, and Hamsaz [23], a tool that analysis a replicated object and establishes an order between conflicting operations and dependencies, proposes a way to decrease it (discussed in detail in Chapter 3).

Also, some states that it's important to combine consistency models, called **hybrid consistency** [6, 8, 27]: the programmer may choose which operations should request weaker or stronger consistency, depending on the consequences of its concurrent execution, i.e., the possibility of leading to inconsistent states.

2.5 CRDTs

Ensuring consistency and availability is difficult, as mentioned in the CAP theorem (see Section 2.3). A highly available system should adopt the eventual consistency model (discussed in Section 2.4) to reduce network latency, executing operations on arrival and propagating their effects in the background. However, it does not guarantee data integrity, since conflicts may occur if problematic operations are executed in concurrency.

It is very difficult for the programmer to reason about problematic concurrent operations. In this manner, Shapiro et al. [52] proposed a data type called **Conflict-free Replicated Data Type (CRDT)**, intended to be used in large-scale distributed applications, in which updates do not require synchronization between replicas to be executed and remain responsive even with potential network failures or high network latency. CRDTs are mutable objects replicated and interconnected by an asynchronous network, which can partition and recover [52].

CRDTs provide, at least, eventual consistency: a set of replicas that have executed the same set of updates eventually reach the same state (converge). Additionally, they provide a stronger notion called strong eventual consistency (SEC) [52], which ensures state convergence and solves conflicts through the use of mathematical properties such as commutativity (e.g., in a replicated counter, the state converges because increment and decrement operations commute [52]) or set theory (e.g., merging sets: merging the sets $\{1,2\}$ and $\{2,3\}$, results in the set $\{1,2,3\}$). Each CRDT provides mainly two kinds of operations: `read` and `update`. They are executed at the source replica, but only the `update` operation needs to be eventually propagated to other replicas since it modifies the object state. Consequently, conflicts may arise, but CRDTs can resolve conflicts by adopting a conflict resolution policy while merging replicas' states. For instance, consider a set of integers, with `addElement` and `removeElement` operations. These operations do not commute when the element is the same, thus conflicting when executed in concurrency. A CRDT can use several concurrency semantics, such as *add-wins* (in which the addition wins over the removal) or *rem-wins* (in which the removal wins over the addition) [52].

Next, two of the most popular types of CRDTs are presented:

- **State-based CRDTs or Convergent Replicated Data Types (CvRDT)** follows the properties of passive replication, described in Section 2.2. This type of CRDT is inefficient for large states, due to the size of the messages that need to be propagated.

- **Operation-based CRDTs or Commutative Replicated Data Types (CmRDT)** follows the properties of active replication, described in Section 2.2. This type of CRDT is useful when all concurrent updates are commutative.

A simple example of the usage of these types of CRDTs is a replicated counter, which can be incremented, decremented and its value queried. If a state-based CRDT is used, then the state has to be propagated by the source replica and merged at receiving replicas. In this scenario, the `merge` operation of the CRDT could be a max function (an increment-only counter was considered), that returns the maximum value between the local state and the remote state. An operation-based CRDT would be much easier to specify since both update operations (increment and decrement) commute. Propagating only the effects of the operations would guarantee to achieve the correct final state.

2.6 Operations Conflicts and Dependencies

A replica, upon a request to execute a single operation, will only continue if the pre-condition is satisfied and the invariant is preserved. When multiple operations are executed concurrently, in a way that each preserves the invariant, the merge of the operations after propagation might break it and, consequently, applications' correctness is compromised. Well, this occurs due to conflicts, which are common in geo-replicated applications that are built on top of weak consistency models.

To help to reason about those events, a set of standard case studies has been used in the literature, such as the courseware application case study. Next, it is shown a description of the specification of that standard example, adopted from Indigo [6] and Hamsaz [23]. This section ends with some examples of conflicts and dependencies, using the provided case study.

2.6.1 Case Study: Courseware

The courseware example [6, 23] has been used to illustrate and assist in understanding how a replicated object behaves when executed in concurrency.

A courseware replicated object is composed as follows:

1. A database state that has three different relations of sets: students, courses and enrollments (between students and courses).

2. An invariant that states that if a specific student or course exists in the enrollment relation, then it should also exist in the students and courses relations, respectively.

3. A set of methods: `register(student)` and `addCourse(course)` to register a new student and course, respectively; `deleteCourse(course)` to delete a course; `enroll(student,course)` to enroll a student in a course; and `query` to obtain object's state [23].

Note that students and enrollments cannot be deleted and operation's arguments are integer identifiers. Table 2.1 shows the requirements and effects of executing a given operation.

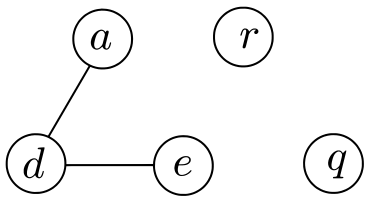


Figure 2.1: Conflict Graph ([23])

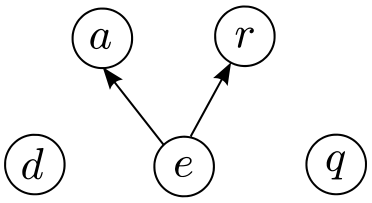


Figure 2.2: Dependency Graph ([23])

Table 2.1: Courseware application specification.

Operation	**Pre-condition**	**Post-condition**
`register(student)`	Id must be non-negative.	The given student exists in the students' set.
`addCourse(course)`	Id must be non-negative.	The given course exists in the courses' set.
`enroll(student,course)`	The given student is registered and the course exists.	A tuple with the given student and course exists in the enrollments' set.
`deleteCourse(course)`	The given course exists and has no students enrolled in it.	The given course no longer exists in the courses' set.

2.6.2 Conflicts

Figure 2.1 presents the conflict graph of the courseware: `addCourse(c)` conflicts with `deleteCourse(c)` and `deleteCourse(c)` conflicts with `enroll(s,c)`.

Consider that the set of students has a student *s* and the set of courses has a course *c*. A request for deleting a course *c* arrives at replica *r1* and another request to enroll a student S in the same course arrives at replica *r2*. The pre-condition in both replicas holds since both student and course exists. If the two requests are executed without synchronization, propagated to other replicas and executed on arrival, the state of the replicas will diverge. Moreover, the invariant is not preserved since the student S will be enrolled in a non-existing course.

Now consider that the set of courses is empty. Replica *r1* receives a request to add a course *c* and executes it; right after, receives another request to delete the course *c* and also executes it. Concurrently, a request to add the course *c* arrives at replica *r2* and it is executed. The pre-conditions in all requests hold. If the effects of the operations executed in *r1* and *r2* are propagated, what happens to the course *c*? A conflict resolution must be defined to decide whether the addition prevails over the removal or vice-versa. CRDTs can be used for this purpose (see Section 2.5).

It is important to emphasize that no conflicts arise if the parameters of the operations are different. For this reason, we can state that the conflict graph is only applied when the added course is the same as the one being concurrently deleted. The same approach applies when enrolling a particular student in a different course than the one being concurrently removed.

2.6.3 Dependencies

In Figure 2.2 is presented the dependency oriented graph of the courseware application, where the execution of the operation `enroll(s,c)` needs that both the course *c* and the student *s* are registered in the system. Commutativity is a property used to verify dependency relations, that is if swapping the order of the operations, results in different outcomes.

Consider three replicas: *r1*, *r2* and *r3*. A request arrives at replica *r1* to perform the enrollment operation: `enroll(s,c)`. Concurrently, replica *r2* receives and executes the effects of a previous operation that was already executed in *r1*: the addition of the course *c*. Later, *r3* receives both operations' effects. If it applies the enrollment before the addition of the course, then data integrity is not preserved, because these operations do not commute. Thus, the effects of `enroll(s,c)` should be applied after the effects of `addCourse(c)`.

2.7 Program Specification

As discussed before, building distributed applications is difficult. Traditional approaches to verify the correctness of an application include monitoring runtime behavior to verify implementation problems (e.g., debugging), writing unit tests (JUnit, for instance) and evaluating tests coverage. These approaches stand for **dynamic verification**, which is done at runtime. However, detecting all possible errors is not straightforward and it might be almost an impossible task to write tests for every possible case scenario.

Alternatively, **static verification** may be used, since it verifies at compile time by analyzing the source code. When writing a program, we need to reason about its state, i.e., assert which conditions are satisfied before the program starts, and assert those in which the final state results. The former assert is called a pre-condition and the latter a post-condition, that can be applied in methods and represented as a Hoare triple [22]:

$$\left\{ Pre \right\}\ Program\ \left\{ Post \right\}$$

Along with them, we need to define class invariants, that behave as global assertions and must remain true throughout the program life cycle, from the moment it is instantiated. This whole idea stands for a correctness methodology, used in object-oriented languages, known as **Design by Contract** [13]. In the context of databases, invariants are the rules that must be maintained, such as data constraints.

The specification of an application is achieved through the use of a specification language, making use of the static verification approach explained earlier, such as Dafny, developed by Microsoft [43], or WhyML, put into practice on Why3 platform [42]. The ambition of specification languages is to prove application correctness, i.e., the implementation is correct regarding a specification.

Static analysis tools, such as **CISE3** [42], analyze the specification of an application and define which operations are problematic under concurrency (i.e. conflicts between operations), as well as their dependencies. Static analysis tools reason on when synchronization should be introduced to preserve application correctness, that is, preserving integrity invariants. The input of the protocol that will be elaborated in this dissertation will receive as input a set of operations' conflicts and their dependencies, provided by these static analysis tools, more specifically CISE3 [42]. Through a parameter analysis, the tool provides the causal relationship between operations. We call *parameter dependency* and *name dependency* to the causal relationship between operations that are related by parameter and name, respectively, which will be explained in detail in Section 4.1.1.

CISE3 [42], a plugin for the Why3 framework [42] implements a proof rule that maintains integrity invariants. The programmer needs to provide the application specification, i.e. the pre and post conditions, as well as the global invariants. The proof is modular, i.e., is able to reason about the behavior of one operation at a time [46]. It assumes causal consistency by default and allows the programmer to specify, using a *conflict relation*, which pairs of operations are problematic, by the means of tokens. For instance, in the bank application example, the conflict relation could be used to require synchronization for any pairs of concurrent withdrawals; a set of tokens would be associated with these withdrawals, preventing from executing without synchronization, relying on strong consistency. If an operation is requested to update a certain resource and another operation already has the token associated, its execution is prevented [46]. The tool contemplates a proof rule with three proof obligations:

1. **Safety analysis:** verifies if any single operation executed without concurrency maintains the invariant.

2. **Commutativity analysis:** checks if every pair of operations commute, i.e., if changing their order of execution leads to the same state.

3. **Stability analysis:** verifies if every precondition of each operation is stable under the effects of all other concurrent operations.

If all rules are satisfied, the invariant is guaranteed to be preserved in every possible execution. If Why3 is not able to prove any assertion in the program, then the correctness is not guaranteed, and it presents a counterexample.

Still related to the specification part, it would be interesting, as future work, to develop a specification of the protocol and its correctness properties in TLA$^+$ [25], using TLC model checker [25] to validate the protocol. TLA$^+$ is a specification language, developed by Leslie Lamport, used to design, model and verify concurrent and distributed systems.

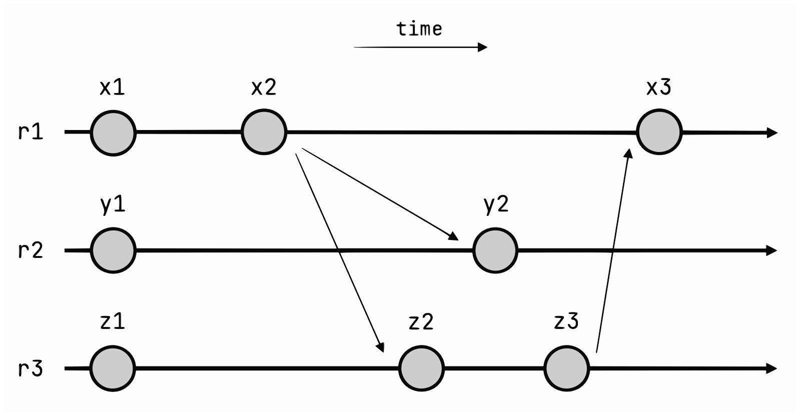


Figure 2.3: Message exchange between three replicas. Circles represent events.

2.8 Causality Tracking

Causality is an essential concept in asynchronous distributed systems, that is, in a set of replicas that do not share the same global memory and that communicate through message passing. Typically, the sending and receiving of messages, as well as the change of state in a replica, are also called events [10]. For any two events, *i* and *j*, it is necessary to recognize whether *i* causally relates to *j* and vice versa. In case *j* is causally dependent on *i*, then it is said that the execution of *j* cannot occur without the execution of *i* [10, 21]. Another property that is verified is that if the events are not causally related, then they are said to be concurrent.

These notions are defined in the ***happened before relationship*** [26], a partial order between events, proposed by Leslie Lamport in 1978. The event *i* can be the cause of event *j*, or as the relationship defines, *i happened before j* (denoted $i \rightarrow j$). If the events *i* and *j* occur in the same replica, then the event *i* is executed first. In case they originate in different replicas, the replica where *j* occurs must previously know the event *i* by another replica. In both cases, we can outline a path from one event to the other. As an example, following Figure 2.3, we can state that there is a path from *x1* to *z2*, i.e., $x1 \rightarrow z2$. Another property that is verified is the transitivity between events: as $x1 \rightarrow z2$ and $z2 \rightarrow x3$, then $x1 \rightarrow x3$. When we can't outline a path, we say that the events are concurrent: $y2 \parallel z3$ or $y2 \parallel x3$, for instance.

Formally, for two events, *i* and *j*, the *happened before relationship* translates into the following conditions[26]:

- If *i* is the cause of *j*, then $i \rightarrow j$;
- If $i \rightarrow j$ and $j \rightarrow$ k, then $i \rightarrow$ k (transitivity);
- If we cannot relate the events, then we say that they are concurrent: $i \nrightarrow j$ and $j \nrightarrow i$, or simply, $i \parallel j$.

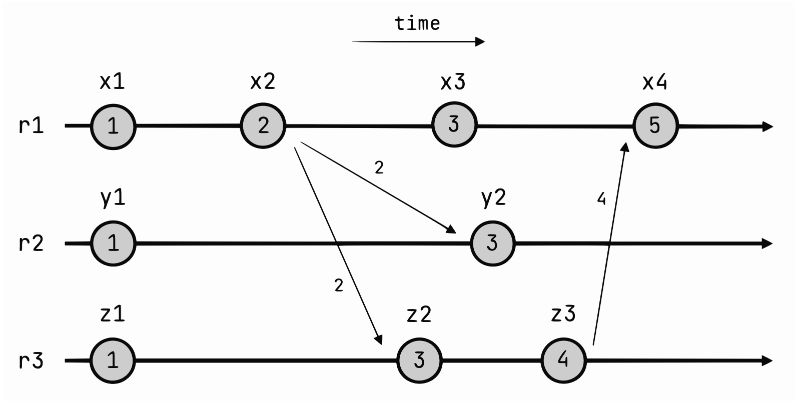


Figure 2.4: Message exchange using Lamport timestamps.

2.8.1 Logical Clocks

Each replica in a distributed system holds its clock, and not all share the same physical clock. In other words, they are not precisely synchronized with each other, and there is no way to sort events using a shared global clock. Therefore, logical clocks are used to sort events and, consequently, as a mechanism for determining causality. They can capture the *happened before relationship* by assigning a timestamp to each event that occurs in a replica. This timestamp is a counter, which corresponds to a non-negative integer, and its value increases monotonically.

Lamport [26] proposed the first logical clock implementation, typically known as **Lamport timestamp**. Each replica holds a local integer variable C_i, the logical clock or counter, which initially has the value 0. When an event occurs:

(a) If it occurs locally (a tick event), the counter is incremented by 1: $C_i = C_i + 1$;

(b) In the case of sending a message, it attaches the value of the counter;

(c) If the event occurs in a replica *j*, a replica *i* upon receiving the message, the counter modifies as follows: $C_i = max(C_i, C_j) + 1$;

Following Figure 2.4, where for each event, we attach the respective timestamp, we can confirm the previous properties: (a) in replica *r1*, from event *x1* to *x2* the counter increased by 1; (b) the propagation of the message includes the attached timestamp; and (c) the counter in replica *r1* results in 5 since $C_{r1}(x4) = max(3,4) + 1 = 5$.

In this algorithm, if $i \rightarrow j$, then $C_r(i) < C_r(j)$. However, the opposite does not hold due to potential concurrent events. In Figure 2.4, $C_{r1}(x3) < C_{r3}(z3)$, however, asserting that *x3* $\rightarrow$ *z3* is false. To satisfy this condition vector clocks were proposed.

2.8.2 Vector Clocks

Vector clocks were initially proposed by Fidge [18] and Mattern [40]. These clocks are widely used to track causality in causally consistent systems.

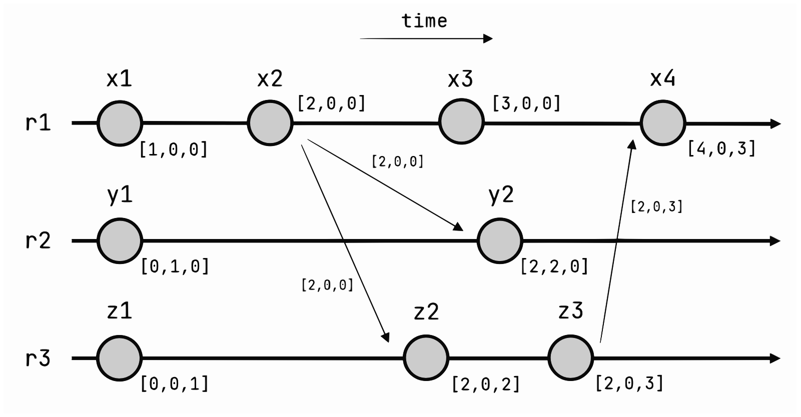


Figure 2.5: Message exchange using classic vector clocks.

Each replica maintains a vector *VC* of the size equal to the number of existing replicas. Each index of this vector, which starts at 0, contains the counter associated with a particular replica. When an event occurs:

(a) If it occurs locally, for a replica i, the counter corresponding to its index in the vector is incremented: $VC_i[i] = VC_i[i] + 1$;

(b) When propagating a message, it attaches the current version of the vector;

(c) Upon receiving a message m, a replica i increments its counter by 1, such as in (a). Also, updates each index in the vector according to what it has received. For each counter, it updates according to the maximum between what it had in its vector and what it received: $VC_i = max(VC_i[k], VC_m[k]), \forall\, k \in 1, ..., n\ \wedge k \neq i$, being n the number of replicas.

Consider two replicas: i and j. When comparing vector clocks, the following properties are defined [48]:

(a) $VC_i \leq VC_j \iff \forall\, x : VC_i[x] \leq VC_j[x]$

(b) $VC_i < VC_j \iff VC_i \leq VC_j\ \wedge\ \exists\, x : VC_i[x] < VC_j[x]$

(c) $VC_i \parallel VC_j \iff \neg(VC_i < VC_j) \wedge \neg(VC_j < VC_i)$

Relating to the *happened before relationship*, given two events x and y [18]:

$$x \rightarrow y \iff VC(x) < VC(y)$$

Figure 2.5 serves as an example to validate the previous properties. For instance:

$$x2 \rightarrow z3 \ \text{ since } \ 0 < 3 \qquad\qquad x4 \nrightarrow z3 \ \text{ since } \ 4 > 2$$

$$x3 \parallel y2 \ \text{ since } \ VC_{r1}(x3)[0] > VC_{r2}(y2)[0] \ \text{ and } \ VC_{r2}(y2)[1] > VC_{r1}(x3)[1]$$

Unlike using Lamport timestamps, where $x3 \rightarrow z3$ because $C_{r1}(x3) < C_{r3}(z3)$, using vector clocks we can verify that $x3 \nrightarrow z3$, i.e., $x3 \parallel z3$ since $VC_{r1}(x3) \nless VC_{r3}(z3)$ and $VC_{r3}(z3) \nless VC_{r1}(x3)$.

However, there are some drawbacks associated with vector clocks. As in Figure 2.5, besides each replica keeps a vector of the size of the existing replicas. Also, it is necessary to broadcast the vector in each message. Well, this vector grows linearly according to the number of replicas, making this approach unfeasible since there is an increased overhead in the network. Some proposals to reduce this overhead in asynchronous systems include increasing the counter only in events considered relevant, such as write operations [1, 24, 48]. Or to broadcast only the clock differences since the last one that was sent to the target replica [21, 48, 50, 53, 54]. However, it is usually necessary to keep the vector size n, being n the number of replicas. Recently, it was proposed an approach that holds the clock as a single prime number [24, 47]. Although only one number is broadcast over the network, which is beneficial, this number grows rapidly, which can cause overflow and requires extra computations to reset the clock.

Chapter 3

Related Work

This chapter mainly discusses different approaches regarding the balance of synchronization and availability in distributed systems. It presents related work on how to handle concurrent conflicting operations and their dependencies.

3.1 Hamsaz

Houshmand et al. [23] implemented a tool called Hamsaz, whose goal is to automatically build a correct replicated object that preserves data integrity and avoids unnecessary synchronization [23].

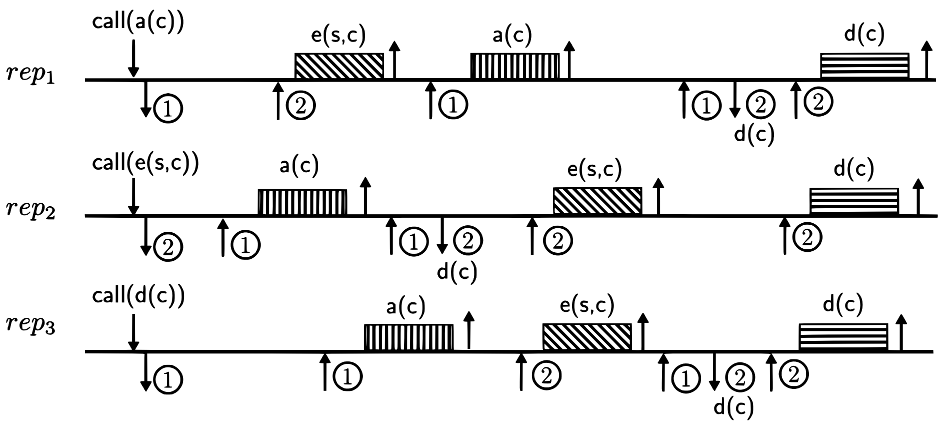
Hamsaz's system model is based on a sufficient condition called *well-coordination*, which states that conflicting and dependent operations require synchronization and causality, respectively. It presents a static analysis that, given the specification of a system, includes an object definition containing the state type and the system's invariant. Then, a SMT solver is used to define which pairs of operations conflict, as well as their dependencies [23]. Then, the results of the static analysis are used to instantiate the protocols: non-blocking or blocking synchronization protocols, that will be explained next. The protocols can be built on systems that use eventual consistency, causal consistency, and strong consistency [23]. Both protocols do not consider operations' dependencies. The courseware case study (described in Section 2.6.1) is used to describe how the protocols operate.

3.1.1 Non-Blocking Protocol

The non-blocking protocol considers that the failure of one replica does not affect the progression of the remaining replicas. It defines two protocols: the total order broadcast (TOB) protocol [56], ensuring that messages are always delivered in a given order in all replicas, and the reliable broadcast, which guarantees that propagated messages are always delivered to remote replicas. The TOB protocol could lead to deadlocks, preventing the system from progressing, so alternatively it is possible to use a variant of the TOB protocol called multi total order broadcast (MTOB). The difference between them is that TOB propagates individual messages, while MTOB propagates sequences of messages.

The implementation of the non-blocking protocol uses cliques of the conflicting graph. A clique is a subset of vertices of a non-directed graph such that every two distinct vertices are adjacent. Operations pertaining to the same clique should synchronize. Using the conflict graph of the courseware application illustrated in Figure 2.1, the cliques are *cl1* = {`deleteCourse`, `addCourse`} and *cl2* = {`deleteCourse`, `enrollment`}. These cliques are also maximal cliques: a clique is maximal if it is not a subset of a larger clique. The operation to delete a course is in multiple cliques, so a call to this operation must be totally ordered with respect to calls for each of those cliques. The call is broadcast to all MTOB instances and executed only when it is ordered and delivered by all of them [23]. Queues are used to this effect, meaning that a call can be executed only when it appears at the head of the queues of all MTOBs that is broadcast to [23].

The non-blocking protocol is instantiated with the set of cliques *Cl*, and each clique *cl* has a queue *q* and an instance of an MTOB object *mtob*.



The symbols ↓ and ↑ show requests to and responses from the protocols. Events to the main protocol are shown above and events to the sub-protocols are shown below the horizontal time line. The symbols ① and ② represent events of the first and second TOB sub-protocols respectively. Blocks show the execution of method calls.

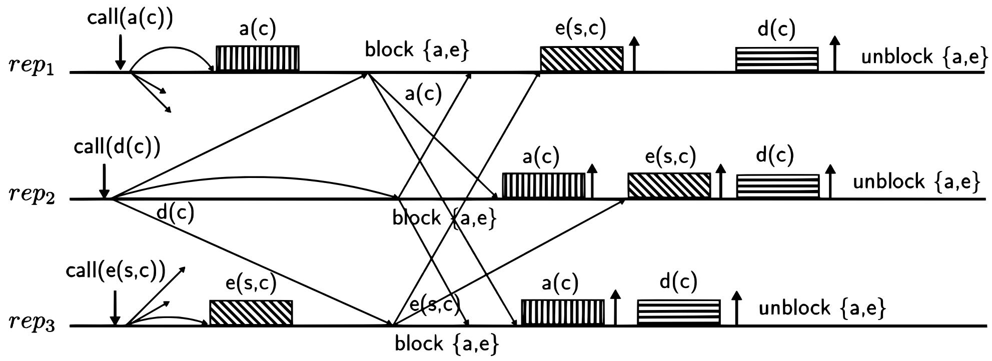
Figure 3.1: Hamsaz non-blocking synchronization protocol ([23]).

Figure 3.1 presents an example with three concurrent executions in replicas *r1*, *r2* and *r3*, respectively: add a course *c*, enroll a student *s* in the course *c* and delete the course *c*. All of this operations belong to a clique, so all are broadcast to its respective clique's *mtob* instance (e.g., call to add the course belongs to *cl1*, so it is broadcast to *mtob1*). Note that in *r3*, Figure 3.1 shows the delete call only broadcasting to *mtob1*, but it broadcasts to both *mtob1* and *mtob2*, since the delete appears in both cliques. Each *mtob* instance decides which order the respective clique operations should be delivered. Since the delete operation appears in several cliques, if it appears first in *mtob1* instance, then it will also appear first in *mtob2* instance. In the example, is considered that *mtob1* first has the addition and *mtob2* first has the enrollment, so the delete operation appears last in both *mtob* instances. Since the addition and the enrollment operations commute, they can be delivered in an arbitrary order, that's why the order in *r1* and *r2* is the opposite. Meanwhile, in *r2*, *mtob1* delivers the addition and the replica executes it. Then, it delivers the delete operation, but since this operation belongs to more than one clique and it is not at the head of the queue *q2* (because *r2* has not yet executed the enrollment operation), it broadcasts to *mtob2* requesting the delete operation. Still in *r2*, *mtob2* delivers first the enrollment, which was at the head of its queue and the replica executes it. Now the delete operation can be executed since it appears at the head of both queues.

3.1.2 Blocking Protocol

Unlike the previous protocol, the blocking protocol prevents the system from progressing when failures arise. Also, not all operations that are adjacent in the courseware conflict graph (illustrated in Figure 2.1) require synchronization.

It starts by calculating the minimum vertex cover of the conflict graph. A vertex cover is a subset of vertices of the graph, such that for every existing edge in the graph, at least one of its endpoints is in the subset. The minimum vertex cover is the smaller-sized vertex cover [23].



The symbols ↓ and ↑ show requests to and responses from the protocols. Diagonal arrows show message transmission.

Figure 3.2: Hamsaz blocking synchronization protocol ([23]).

In the courseware conflict graph, the minimum vertex cover is the delete operation, being the only operation that requires synchronization to be executed. The replica where the call to delete is requested acts as if it was a leader since if it fails, stops other replicas from progressing. A call to the delete method requires that the replicas block, waiting for the operations that are adjacent in the minimum vertex cover to be executed (add and enrollment operations). When replicas are blocked, they broadcast previous updates so the system can progress. When the delete operation is finally executed, the replicas are no longer blocked.

Figure 3.2 presents a similar example to the one seen in the non-blocking protocol, as it has the same number of replicas and the operations executed in concurrency are also the same. In replicas *r1* and *r3*, operations are directly executed and broadcast. Following what was explained earlier, in replica *r2* the delete operation belongs to the minimum vertex cover, so before it can be executed in the replicas, they have to execute the add and enrollment operations before the delete. After the delete is executed, the replicas are unblocked.

3.1.3 Dependency Tacking Protocol

In both non-blocking and blocking protocols, method calls were assumed to be independent [23]. The dependency-tacking protocol is used to track dependencies. It states that if a method call has dependencies, "they should be tracked at the originating node and broadcast together with the call" [23]. It requires causality: receiving replicas should apply the call only after its dependencies are applied.

For ensuring dependencies between methods, conflicting pairs of methods can be ignored, since every pair of conflicting calls have the same order across replicas, so their dependencies are implicitly preserved [23]. It uses a protocol called *inverse atomic protocol*, where replicas have to agree on whether a call to a method can be committed. The decision is to abort if all replicas vote for abort, or commit otherwise. If a replica decides that the method's call is *permissible* (i.e., the invariant is preserved before the method's execution and is also preserved after it) then it votes to commit, along with the dependencies of that call. If any replica receives the abort decision, the call is aborted; otherwise, it waits for the dependent methods to arrive [23].

3.2 Indigo

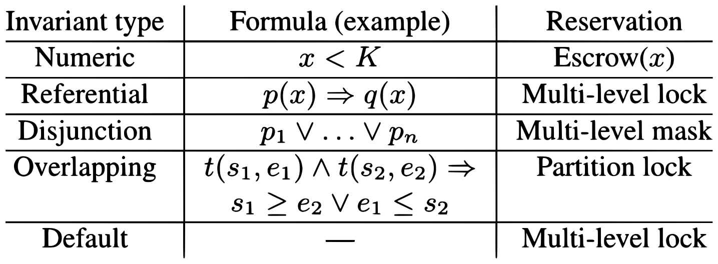
Balegas et al. [6] developed Indigo, a middleware system that supports explicit consistency. In explicit consistency, the programmer specifies application invariants, allowing different replicas to reorder the execution of operations (as long as the invariants are preserved), and also ensures that every state transition preserves the invariant [6]. It builds on a causally consistent storage system and it is based on the following three-step methodology:

1. Performs a static analysis to identify which operations can break the invariant when executed concurrently. It is used an algorithm divided into three functions: (i) *self-conflicting*: operations that conflict with themselves (using the same or different arguments); (ii) *opposing*: if the operations have opposite post-conditions; and (iii) *conflict*: if the operations break the invariant.

2. The programmer chooses between two techniques to coordinate conflicting operations efficiently: invariant-repair or violation-avoidance. The invariant-repair states that conflicting operations are allowed to execute and then conflict resolution policies are applied, such as *add-wins* or *rem-wins*, relying on CRDTs (see Section 2.5). The violation-avoidance technique extends reservations approaches, such as warranties [38], by restricting concurrency sufficiently not to violate the invariant. Summarily, a replica synchronizes to pre-allocate the permission to execute a set of future updates without synchronization, thanks to the reservation [6].

3. The application code is adapted to have calls to the middleware library, in order to use one of the mechanisms chosen in the previous step.

The violation-avoidance technique considers a range of reservations, as illustrated in Figure 3.3. A certain reservation is associated for each invariant type. Indigo maintains an instance for each reservation type, which includes the information about the rights assigned to each replica.



Invariant type	Formula (example)	Reservation
Numeric	$x < K$	Escrow(x)
Referential	$p(x) \Rightarrow q(x)$	Multi-level lock
Disjunction	$p_1 \vee \ldots \vee p_n$	Multi-level mask
Overlapping	$t(s_1, e_1) \wedge t(s_2, e_2) \Rightarrow s_1 \geq e_2 \vee e_1 \leq s_2$	Partition lock
Default	—	Multi-level lock

Figure 3.3: Default mapping from invariants to reservations ([6])

The multi-level lock reservation provides the following rights: (i) *shared forbid* (gives the shared right to prevent some operation to occur); (ii) *shared allow* (gives the shared right to allow some operation to occur); (iii) *exclusive allow* (gives the exclusive right to execute some operation) [6]. When a replica has one of these rights, no other replica may contain rights of another type (forbid or allow). The *exclusive allow* right is useful when an operation conflicts with itself. A replica can give its right to another replica. In case of being the only replica with rights, it can change the type of right and give it to itself or to another replica. In addition, a right can be revoked from the local replica [6]. In the courseware example (described in Section 2.6.1), considering an execution with concurrent `enroll` and `deleteCourse` operations: to execute the delete it is necessary to obtain the *shared allow* right on the reservation for `deleteCourse`; to execute the enrollment it is necessary to obtain the *shared forbid* on the reservation for `deleteCourse`.

The multi-level mask reservation it is used for a disjunction invariant. When a single condition of the disjunction invariant obtains a *shared allow* right, one of the other conditions should obtain a *shared forbid* right [6].

The escrow reservation it is used for numeric invariants of the form $x \geq k$. It allows $x - k$ rights to decrement without synchronization, which can be divided among the replicas. If a replica requests to decrement *n* times then *n* rights are required; if a replica does not have enough rights then the system will try to obtain them from other replicas; if it is not possible then the operation fails [6].

The partition lock reservation "allows a replica to obtain an *exclusive lock* on a interval of real values. Replicas can obtain locks on multiple intervals" [6], but no two intervals overlap.

3.3 IPA

IPA [5] presents an approach to preserve application invariants, under weak consistency, without synchronization on the execution of concurrent updates. Proposes a methodology called compensations, intended to modify applications to automatically prevent invariant violations, through touch operations [5]. Compensations are only used for numeric invariants (e.g., the total number of flight tickets can not be exceeded). The approach uses conflict resolution policies to modify operations' semantics. Moreover, it assumes that the programmer is able to choose a conflict resolution policy, relying on CRDTs [5] (described in Section 2.5). The touch operation does not require synchronization during the execution of operations, and if no conflicting operations are executed, the additional (touch) operations have no observable effect on the database state [5]. To help in the process of detecting problematic concurrent executions, a static analysis is used. Additionally, it is also used to search for modifications that prevent invariant violations [5]. In some cases, it is difficult to prevent having an observable effect. For instance, it is not possible to prevent flight overbooking [5].

In general, IPA presents a conflict repair approach that changes the semantics of the operations. Operations are propagated with extra actions if concurrent updates do not preserve the system's invariant. For instance, in the courseware application (described in Section 2.6.1), an example of a problematic concurrent execution is the enrollment of a student *s* in a course *c* and the removal of the course *c*. If replicas do not synchronize, the invariant is violated. Using the conflict repair, the invariant violation can be repaired after it is detected, by either (i) removing the new enrollment; or (ii) restoring the course *c*, depending on the conflict resolution adopted. If the chosen repair involves restoring the course, then the operation that removes the course is propagated with an extra action: the removal does not take effect.

3.4 Predictive Treaties

Magrino et al. [39] proposed a mechanism called predictive treaties, that guarantees strong consistency with reduced synchronization by predicting future updates based on previous ones. It follows previous work on warranties [38] and homeostasis [49], in a way that increases performance.

Predictive treaties predict computations based on logical predicates over system state [39]. For instance, for a deterministic computation $f(x)$ that produces a value y, the corresponding predicate $f(x) = y$ holds. Also, as long as x remains the same, the value of y can be cached and reused, since the result is always the same [39]. The approach uses estimations to determine until when a predicate will be ensured. In this manner, metrics are used to measure the "distance" until the predicate is no longer preserved and predicts how this distance will change [39]. The metrics allow the predictive treaties to be:

- Time-dependent, reasoning on how the system state changes with time. For instance, if the function $f(x)$ calculates the amount of stock in a warehouse, a predictive treaty might guarantee the inequality $f(x) > 100 - 5t$, where t represents elapsed time in minutes [39].

- Hierarchical, where lower-level treaties may be grouped to imply a higher-level treaty [39]. When a low-level treaty is set in the local state of a replica, local updates do not need to be synchronized, not invalidating local treaties. Thus, higher-level treaties can be applied locally without synchronization. Using a hierarchical structure, distributed synchronization tends to involve a certain group of nodes [39].

It is costly to create and maintain objects that represent metrics and predicates. To reduce them, a mechanism called stipulated commit was introduced, which allows the programmer to propose updates, which are only applied if they do not violate any treaty.

Predictive treaties are useful in applications such as the voting application example [39]. In this example, there are two candidates, *a* and *b* run to win the election; there are two voting stations, nodes *s1* and *s2*, which receive calls for the operations `vote(a)`, `vote(b)` and `winning_candidate()`. Voting operations increments by one value the total votes. At any moment, every node should know who is the winning candidate. For instance, if the candidate *a* is winning by a considerable number of votes, the nodes do not need synchronization to know which candidate is winning. However, if the number of votes of both candidates is close, then synchronization is necessary. A predictive treaty defines predicates and metrics to decide whether the number of votes in both candidates are near to one another: the system takes into account the minimum number of votes that can invalidate the global predicate: if the candidate *a* is the winning candidate, the global predicate is $(a1 - b1) + (a2 - b2) > 0$, being $a1$ the total number of votes on candidate *a* in node *s1*.

3.5 Join Model

In an asynchronous distributed system, it is difficult to reason and implement a dynamic environment where new replicas or nodes join a cluster.

Bauwens et al. [11] define a join model in which replicas dynamically join the cluster. They use operation-based CRDTs built on top of a Reliable Causal Broadcast middleware [14], adapting the notion of causal stability of Baquero et al. [9]. Note that an operation is stable if and only if all replicas acknowledge that operation.

They assume (i) a correct state of the replicas before receiving any join request, (ii) if failures arise, the replicas eventually recover, (iii) there is no loss or duplication of messages, and (iv) the middleware uses acks to ensure message processing.

Regarding the join model [11], initially, the new replica *N* sends a join request to one of the existing replicas in the cluster. Suppose that the replica that receives the join request has an identifier *J*. Replica *J* is responsible for accepting the request, providing *N* with its knowledge of the cluster. This knowledge includes the location of each existing replica. Then, replica *N* connects with the other replicas and adds each one of them to its list of known replicas. However, *N* is already able to start receiving messages, which will have to buffer and will only be able to apply them after receiving a replicated state from *J*. Replica *N* requests the state to *J* after connecting to all replicas and, as soon as it receives the state, *N* can apply the messages that were buffered.

They also discuss how they manage concurrent joins. Consider that two replicas, *N1* and *N2*, intend to join the cluster. They perform the same procedure just described. As they join concurrently, *N1* may not be aware that *N2* also wants to join, and vice versa. For this purpose, one of the existing replicas has to interconnect them. Only after having full knowledge of the network they can request the state. It is said that a replica is prepared when it has applied a replicated state and is connected to all other replicas.

Lastly, they present their approach regarding causal stability in a dynamic environment:

- The replica *J* receives the join request and sends the state to the new replica *N*. If this state contains a stable operation that originated in *J*, then as it is already stable does not need further treatment. If it contains an operation that is not stable, then *J* has to wait for *N* to inform that it has already applied the operation, so that *N* can contribute to the stability.

- For operations that originate in *N*, if *N* is still not prepared, then it buffers the operations. As soon as it receives and applies the replicated state, then it can treat the operations as if it were a "regular node".

- Finally, in the case of operations that originate neither in replica *J* or *N* but in a replica *A*, if *A* already knows that *N* exists, then *A* has to wait for *N* to send an ack to stabilize the operation. If replica *A* is unaware of *N*, the join request from *N* to *J* and the operation that originated in *A* are concurrent. In this case, the cluster has to synchronize so that *A* recognizes that *N* exists. For this, *J* has to inform *A* about the new replica. Then the operation will be included in the state that *J* will send to *N*.

3.6 Conclusions

In this section, we will present a brief conclusion regarding the related work discussed.

Predictive treaties involve an extremely complex process, as the definition of treaties includes the definition of metrics and predicates. IPA has no synchronization, so it uses eventual consistency, but the associated cost is that it changes the semantics of the operations since operations are no longer what the programmer has defined because they are propagated with extra actions. Indigo relies on causal consistency, which demands synchronization effort. In Hamsaz, the blocking protocol blocks replicas, as they wait for certain operations to arrive and exploit synchronization. On the other hand, the non-blocking protocol, despite ensuring the progression of the system, relies on an abstract layer to preserve data integrity, ending up having a global synchronization point. Hamsaz relies on a hybrid consistency model if we consider both protocols. The blocking protocol stands on stronger notions where replicas need synchronization, while the non-blocking protocol relies on weaker notions, such as causal consistency.

Regarding the join model, it has several similarities with what we will present in Chapter 5. However, replicas are not required to know the location of all other replicas in the network since it would be necessary to maintain a data structure with the identification of all replicas. Consequently, we abstract the fact that the new replica has to request its joining and the state to a particular replica. Another aspect is related to persistence, where they only mention the internal state of the replica. Well, having the data persisted in a database helps us to avoid attaching stable operations in the state that is sent to the new replica. In the internal state, we only keep information about the operations until they stabilize.

Chapter 4

Design and Implementation

The following chapter focuses on the developed algorithm, in which numerous iterations have been carried out, from a sequential implementation to a thread-based approach using persistence, relying on Akka [29].

It covers a thorough presentation of the algorithm design. First of all, we will provide a detailed description of the core algorithm, which addresses the most relevant functions. Next, we will describe and illustrate the internal architecture of the Akka modules we adopted. Finally, we present an overview of our architecture, which is the support to comprehend Chapter 5.

4.1 Core Algorithm

The algorithm behind **Semantic Consistency** requires a dependency graph, which is built in advance using the output from CISE3 [42], a static analysis tool, previously addressed in Section 1.3. The static analysis of the client application provides dependencies among operations, including parameter analysis. A client application provides a set of operations or methods. These contain a list of parameters or no parameters at all. As part of the algorithm, we consider a cluster, i.e. a group of replicas, which communicate reliably with no message loss or duplication, and a replica does not fail.

Hereafter, the algorithm description is done together with the aid of Algorithm 1.

We defined some entities, namely a replica, an operation, and an origin. Each **operation** includes: (i) an **origin**, which is composed of the replica identifier and the counter that the replica had while creating the operation; this field uniquely identifies the operation; (ii) a name; (iii) a list of parameters, which can have arbitrary size and each parameter can be of any type of object; (iv) and a set of dependencies related to the origins on which the operation is dependent. Each **replica** has an identifier and an operation counter. Additionally, it contains three maps (lines 1-6 of Algorithm 1):

1. Executed operations, which associates the origin with the corresponding executed operation;
2. Pending operations, that track operations not executed, due to lack of dependencies;
3. And the mapping between operations names and their respective location in the first map, which will help to build dependencies efficiently.

Algorithm 1 Core Algorithm – main functions

```
 1: state:
 2:     replicaId
 3:     counter ∈ $N$
 4:     executedOps ∈ $Origin \rightarrow Operation$
 5:     pendingOps ∈ $Origin \rightarrow Set(Operation)$
 6:     opNamesMapping ∈ $String \rightarrow Set(Origin)$

 7: upon init:
 8:     counter ← 0
 9:     executedOps ← {}
10:     pendingOps ← {}
11:     opNamesMapping ← {}

12: function EXECUTE_LOCAL(opName, params)              ▷ operation name and parameters
13:     deps ← BUILD_DEPENDENCIES(opName, params)
14:     execute(opName, params)               ▷ execute operation with given name and parameters
15:     counter++
16:     origin ← Origin(id, counter)
17:     op ← Operation(opName, params, origin, deps)
18:     executedOps ← executedOps ∪ {⟨origin, op⟩}
19:     opNamesMapping[opName] ← opNamesMapping[opName] ∪ {origin}                ▷
    if list is empty, create new one
20:     broadcast(op)

21: function RECEIVE_OP(op) ▷ an operation contains: origin, name, parameters and dependencies
22:     safeToExecute ← true
23:     for all origin ∈ op.deps do               ▷ find which dependency needs to be executed, if any
24:         if not alreadyExecuted(origin) then
25:             safeToExecute ← false      ▷ not safe; has to wait for an operation from origin to execute op
26:             pendingOps ← pendingOps ∪ {⟨origin, op⟩}
27:     if safeToExecute then
28:         EXECUTE_REMOTE(op)
29:         CHECK_PENDING_OPS(op.origin)

30: function EXECUTE_REMOTE(op)
```

```
31:     execute(op.name, op.params)        ▷ execute operation with given name and parameters
32:     counter ← max(op.origin.counter, counter) + 1 ▷ update counter according to Lamport Timestamp
33:     executedOps ← executedOps ∪ {⟨op.origin, op⟩}
34:     opNamesMapping[opName] ← opNamesMapping[opName] ∪ {op.origin}        ▷
    if list is empty, create new one
35:     broadcast(op)

36: function CHECK_PENDING_OPS(origin)
37:     pendings ← pendingOps[origin]
38:     pendingOps ← pendingOps \ { origin }
39:     for all op ∈ pendings do        ▷ try to execute all pending operations waiting for origin
40:         if op.origin ∉ pendingOps then    ▷ safe to execute if no other pending operation is waiting for origin
41:             EXECUTE_REMOTE(op)
42:             CHECK_PENDING_OPS(op.origin)

43: procedure BUILD_DEPENDENCIES(opName, params)
44:     deps ← {}
45:     succs ← DependencyGraph.predecessors(opName)
46:     for all edge ∈ succs do      ▷ an edge holds a list of pairs regarding the relation among parameters
47:         opNameToSearch ← edge.destination.id                                  ▷
    an edge connects two nodes, which are operations
48:         setOfOrigins ← opNamesMapping[opNameToSearch]

49:         if | edge.index | == 0 then                                    ▷ extract origins
50:             deps ← deps ∪ setOfOrigins
51:         else                                                  ▷ there are params to work with
52:             for all pair ∈ edge.index do                   ▷ index refers to the index in the list
53:                 paramToSearch ← params[pair.first]
54:                 found ← false

55:                 for all origin ∈ setOfOrigins do
56:                     op ← executedOps[origin]
57:                     if paramToSearch == op.params[pair.second] then
58:                         deps ← deps ∪ { origin }
59:                         found ← true

60:                 if ¬ found then                   ▷ throw error if some dependency is missing
61:                     Throw_Error_Msg("Cannot execute local operation")
62:     return deps
```

Upon initialization, the counter starts at 0, and the maps are empty ($L7-12$).

The counter gets updated according to the Lamport timestamp (see Section 2.8.1). In case of executing a local operation, the counter is incremented by one ($L16$); Otherwise, if the replica receives a remote operation, the counter is set to the maximum value between the local counter and the counter from the operation received (by accessing `origin` field), plus one ($L33$).

When a client application requests the execution of operation with a specified name and a list of parameters ($L13-21$), the algorithm starts by building its dependencies using the dependency graph (which will be explained in detail in the next section). It verifies whether the execution is valid. If any dependent operation is absent, it rejects the execution. In case it is validated, it increments the counter and adds the operation to `executedOps` and `opNamesMappings`. And finally, broadcasts the operation.

Upon receiving an operation from a remote replica ($L22-30$), the initial step is to ensure that it meets its dependencies ($L24-27$). Assuming it meets ($L28-30$), the execution proceeds ($L31-36$) and checks whether there is any pending operation to be executed, regarding operation origin ($L37-43$). The verification of pending operations is only performed in case the replica receives remote operations. A pending operation waits for dependencies that have not yet arrived. For instance, `enroll(s1,c1)` is pending in the replica since student `s1` has not yet arrived. As soon as the student arrives, the algorithm triggers the execution of the enrollment. Note that 1) the replica executes all operations to which the student `s1` is the only dependency left to arrive; 2) the student `s1` exists in the cluster (i.e., all replicas have student `s1` in their internal state), and the enrollment is sound since at least some other replica already has it on its internal state. Also, it is noteworthy to emphasize the recursive call on line 43: the arrival of an operation that was pending might trigger multiple executions of other pending operations.

4.1.1 Dependencies Construction

Next, we present the dependency graph design, which is the key to understand how we build dependencies, as well as a brief description following the algorithm.

4.1.1.1 Graph Construction

Section 2.6 depicts the graph of dependencies of the courseware application. We mentioned that the enrollment operation is not dependent on all student and course registrations. Therefore, we can gather the results of the static analysis tool CISE3 [42], which provides the dependencies between operations, including parameter analysis, and build the graph appropriately.

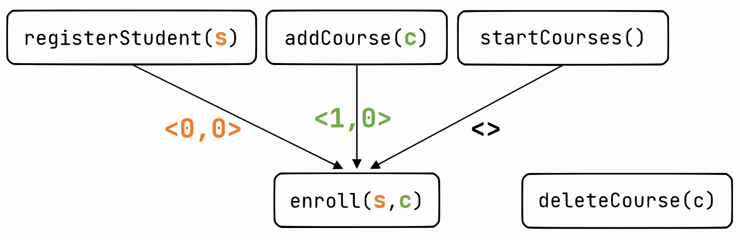


Figure 4.1: Courseware graph construction.

As shown in Figure 4.1, the dependency relationship between `enroll` and `addCourse` consists of the parameter with index 1 of the former operation and index 0 of the latter.

In case there is no pair on the edge label, the operation name drives the relationship. For instance, `enroll` would be dependent on all `startCourses` that had occurred. Note that `startCourses` is a hypothetical operation to present the case when the *parameter dependency* does not apply.

Resorting to graph theory, we say that operation `A` is a predecessor of operation `B` if there is an edge from `A` to `B`. In operations `A` and `B`, the edge is called the outgoing and incoming edge, respectively. By our graph construction, we say that `addCourse` is a predecessor of `enroll`. In terms of execution, `enroll` can only execute if the replica has already executed all its predecessors.

To generalize, the construction of the new graph involves adding information on the edges labels, namely the relationship between parameters, represented as a list of pairs $(<i,j>,<k,m>,\dots)$. The first integer corresponds to the parameter of the operation that has the incoming edge, and the second to the parameter of the operation on the outgoing edge.

4.1.1.2 Build Dependencies

The construction of causal dependencies is a decisive step in the execution of local operations. The replica prevents the execution in the absence of any dependency, guaranteeing the preservation of application invariants.

Following the Algorithm 1 as from line 44, the procedure receives an operation name and parameters and returns a set of origins. A call to the dependency graph is executed, returning a set of operation's predecessors ($L46$). These predecessors correspond to the incoming edges, as illustrated in Figure 4.1. For each edge, if it lacks information, then the existing relationship behaves as a *name dependency*. Thus, it stores all origins regarding the predecessor ($L50-51$). Otherwise, it is a *parameter dependency* and assesses each of the pairs ($L53-60$). For instance, following Figure 4.1, `enroll` has three predecessors, which are its causal dependencies. If it lacks the `addCourse` with the same course (*parameter dependency* between parameter 1 of `enroll` and parameter 0 of `addCourse`), the replica prevents its execution. As long as the replica state contains an operation with the evaluated parameter, it complies with the dependency and proceeds to the next one.

4.2 Architecture

In this section, we will discuss the architectural details of the implemented synchronization protocol. We used the Scala programming language [1], coupled with the use of the Akka actor-based toolkit [29].

[1]See https://www.scala-lang.org/

4.2.1 Akka

In an early stage, we started by implementing the algorithm sequentially, using Scala. [1] Also, we have implemented some frequently used case studies in the literature: the courseware described in Section 2.6.1, auctions [20, 23, 45], payroll [3, 23], and non-negative counter [23, 51]. These case studies supported the verification and execution of the algorithm. Eventually, we knew that we would have to adopt a thread-based implementation, which would be closer to a real distributed application.

Multi-core CPUs are widespread, so we need to benefit from such processing power to optimize the performance of our applications. Furthermore, a distributed application should hold a considerable amount of users performing multiple read and write operations simultaneously. For that purpose, distributed systems run in a multi-machine environment, operating simultaneously and sharing resources. Several processes can run concurrently using threads, so it is essential to have concurrency control to avoid losing updates or have inconsistent retrievals [12]. However, in practice, concurrency control might attend extra concerns. For instance, the use of threads results in the need to acquire locks to access and update shared variables. It may also be necessary to use concurrent data structures, such as `ConcurrentHashMap` or `BlockingQueue`. In a distributed environment, where several machines have to coordinate, one could adopt distributed locks to preserve integrity invariants, which is challenging to scale and presents high latency due to intensive message interchange [37].

The **actor model** is a widely-used programming model that deals with concurrency issues. It eschews the concern of applying low-level code, making the programmer more focused on the main algorithm. The model is composed of a set of actors who exchange messages asynchronously and cannot invoke each other's methods or modify their fields.

Akka is an implementation of the actor model, which provides a set of open-source libraries that allows the development of distributed and concurrent applications on the *Java Virtual Machine* (JVM). Akka supports the usage of Scala since Scala compiles down to JVM code.

As Figure 4.2 depicts, the following components form an actor in Akka:

- **ActorRef** - It forms the logical address or reference of the actor, whose purpose is message passing. The ActorRef hides the actor's instance, preventing access to its methods and fields. Hence, the only way to communicate with an actor is by sending messages;

- **Mailbox** - A message queue;

- **Dispatcher** - Handles enqueueing to and dequeuing messages from the mailbox. Upon dequeuing, messages are processed by the actor, one at a time. It can also schedule when to dequeue, for instance, to handle a message periodically.

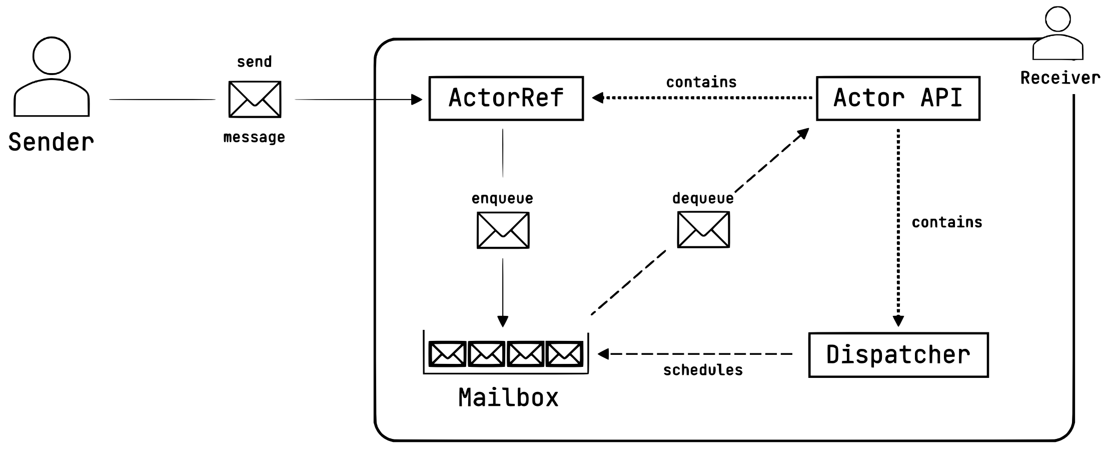


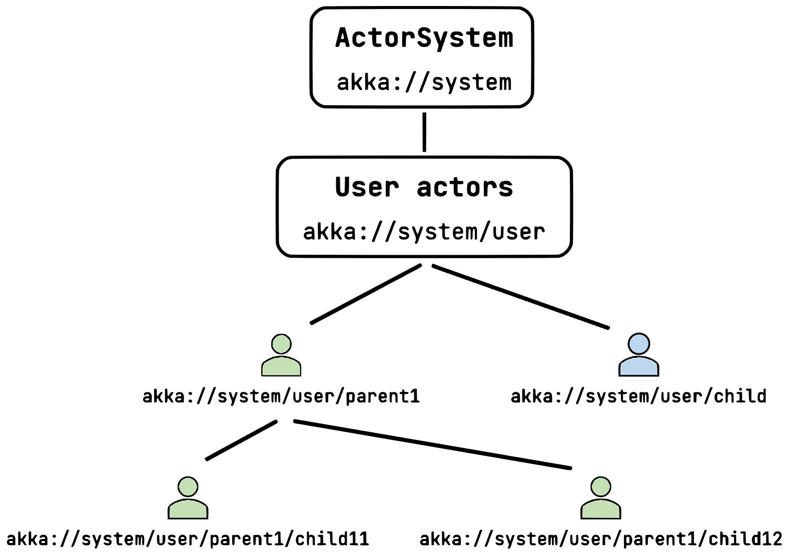
Figure 4.2: Akka actor architecture.

- **Actor API** - The component we have to implement. It defines the actor's behavior when processing a message, which typically includes access and modification of the actor's internal state (data structures, for instance).

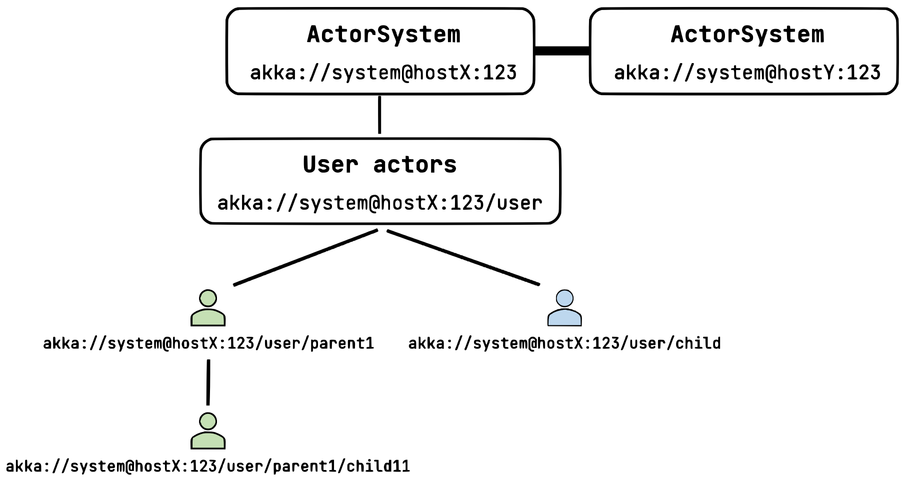
Each actor belongs to an ActorSystem: a hierarchical structure of actors organized as a file directory:

- akka://systemName/user/parent/child, if it has actors on the local system, illustrated in Figure 4.3 (a);
- akka://systemName@127.0.0.1:123/user/parent/child, in case of actors being remote, where 127.0.0.1:123 corresponds to the IP address localhost on port 123. Figure 4.3 (b) depicts an example of a remote hierarchical structure.

In Akka, actors have a life cycle, which comprises their creation to their termination. An actor termination is related to, for example, an exception thrown or a termination message from the actor's parent. Despite being terminated, Akka keeps the actor's path, which is the name of its logical address or reference. When it is started again or restarted, it creates a new instance of the actor and reuses the actor path [28]. Also, there is a special actor called `deadletters` to which messages whose receiver is an actor who has terminated or does not exist are delivered [35].


ActorSystem
akka://system
User actors
akka://system/user
akka://system/user/parent1
akka://system/user/child
akka://system/user/parent1/child11
akka://system/user/parent1/child12

(a) Local actor system.


ActorSystem
akka://system@hostX:123
ActorSystem
akka://system@hostY:123
User actors
akka://system@hostX:123/user
akka://system@hostX:123/user/parent1
akka://system@hostX:123/user/child
akka://system@hostX:123/user/parent1/child11

(b) Remote actor systems.

Figure 4.3: Hierarchical structure of actor systems in Akka.

4.2.1.1 Cluster Publish/Subscribe

Akka provides numerous modules. Two of them are Akka Remoting and Akka Cluster, whereas the latter includes the former. Thus, the actors belonging to the cluster are remote.

Akka Cluster provides the concept of "membership" in which there is a group of nodes that know each other, i.e., that know the location of their neighbors on the network [33]. They establish communication and keep track of the status of each of their neighbors, whether or not it is achievable, using the gossip protocol, based on a failure detection mechanism [34]. Amazon's Dynamo system [16] is the basis for the cluster membership concept in Akka [34]. Moreover, the cluster contains a leader and seed nodes [34]. The leader is usually the oldest node and responsible for propagating state changes to the others, such as member joining and leaving. Seed nodes are the ones to which new nodes communicate to join the cluster.

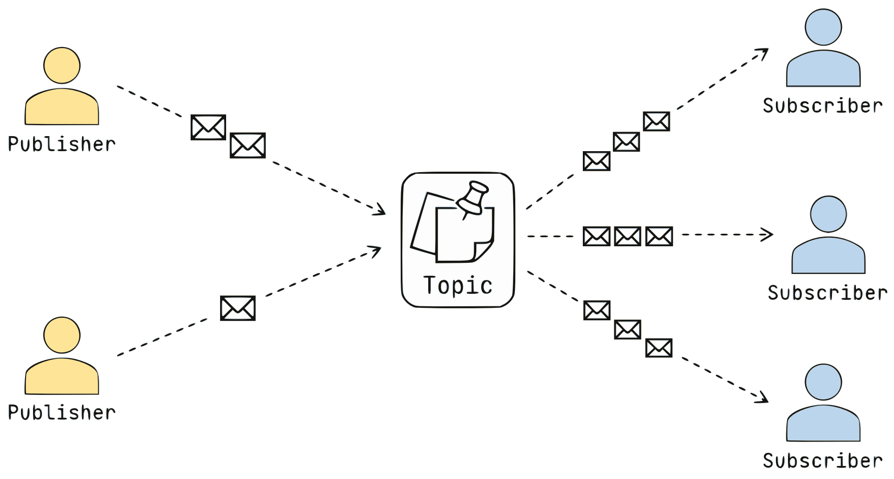


Figure 4.4: Topic-based publish/subscribe architecture.

Over the Akka Cluster core environment, Akka also provides the implementation of a topic-based publish/subscribe paradigm [31]. Publish/subscribe [17], or pub/sub, is an architectural pattern that operates as a messaging middleware, as illustrated in Figure 4.4. It is used in highly available systems, providing a loosely coupled manner of interaction [17]. It focuses on the broadcast of messages asynchronously between publishers and subscribers regarding a topic. The events are produced independently of the subscribers. Subscribers express interest in a given topic and receive notifications whenever there is a publication involving it. Publishers are responsible for producing or publishing messages. The architecture allows decoupling the sender from the receivers, i.e., subscribers and publishers do not need to know the origin of the messages and who will use those messages, respectively. Also, we consider that every replica actor in the cluster can act as a publisher and a subscriber. Regarding the pub/sub abstraction, the programmer defines which messages are published and received on a given topic while implementing the Actor API.

Internally, Akka uses an actor called `DistributedPubSubMediator`, who acts as a mediator in the cluster, in which the established communication is eventually consistent [31]. It is responsible for message passing from and to all other mediators. In our case, all replicas initialize the mediator actor.

Our implementation benefits from the decoupling of actors in the cluster, in particular, to determine the stability of operations. Typically, tracking causal stability implies the usage of vector clocks, as described in Section 2.8. Each node or replica stores a vector with the size of the existing replicas. Though there are techniques to prevent the transmission of the entire vector over the network (see Section 2.8.2), the size of the broadcasted messages grows according to the replicas size. On behalf of the algorithm, we do not want to keep the reference of all nodes, neither an array with length equals the size of existing replicas. We only need to keep an integer matching the size of the cluster, which will be presented in detail in Chapter 5.

4.2.1.2 Akka Persistence

Another module that Akka features is Akka Persistence, which allows persisting data. An actor ceases to be a regular actor and becomes a persistent actor, which has an internal state and persisted data associated.

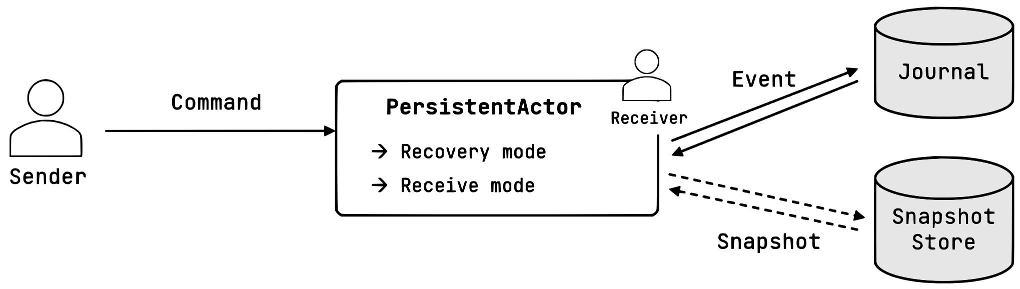


Figure 4.5: Akka persistence architecture.

As shown in Figure 4.5, a persistent actor receives commands and events. The former corresponds to a message sent by other actors, and the latter associates a single state change. If an actor starts and persists all its internal state changes, then the list of events leads to the current state [32]. The entity that stores the events is called journal. Besides, there is also another store, the snapshot store, which keeps the snapshots. A snapshot stores the actor's state at a particular moment.

Whenever an actor starts or restarts, initiates in a recovery mode, in which it receives the last persisted snapshot and subsequent events, so that it can recover its state. In Figure 4.6, let's consider that the snapshot is unique, and it keeps the state at the moment right after the register of the event 100. It is unnecessary to reproduce previous events, but those after it. Once the recovery completes, the actor proceeds to the receiving mode, in which it can receive and process commands.

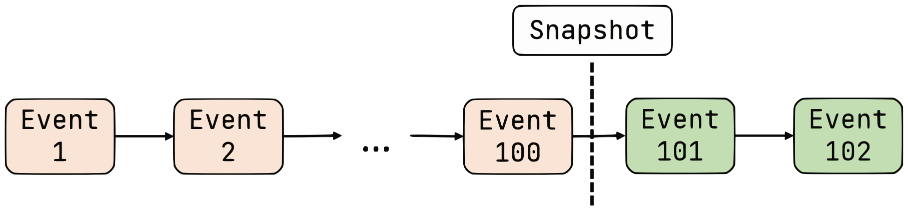


Figure 4.6: Recovery mode: snapshot as a set of events.

By default, Akka uses the local file system through a plugin for a LevelDB [2] instance. This plugin only provides the journal, and it is not suitable to use in conjunction with an Akka cluster since it persists on local storage [36]. Nevertheless, other plugins provide both stores, such as the Akka Persistence Cassandra plugin [30]. We chose to include Apache Cassandra [3] in our architecture since it's a distributed NoSQL database that is suitable in environments that require high performance, availability, and scalability. Cassandra belongs to the NoSQL database group, where one designs the tables according to queries. It has its SQL-like language, called Cassandra Query Language [4] (CQL), in which there is no join mechanism between tables or subqueries, unlike traditional relational SQL databases.

We designed and implemented the data model for each case study in which we decided to evaluate our algorithm. In addition to the creation of the tables regarding the case study (or client application), we also maintain another table related to stable operations.

4.2.2 Summary

To summarize, and following Figure 4.7, there is a set of replicas, which form a cluster. The cluster implementation uses the pattern publish/subscribe based on topics. Each replica, which is a persistent actor, is the parent of two other actors: a listener who listens for state changes of the nodes that pertain to the cluster, such as if a member is running and stops or is leaving the cluster, and a mediator who publishes messages and receives notifications about the topics of interest. The replica has an internal state and persistent data, whereas, for the latter, we are using a connection to the NoSQL Cassandra database.

In the illustration below, solid arrows represent messages exchange. Note that the outgoing arrow from the listener and the mediator to the main actor represents communications that inform the parent. Similarly, the replica sends a message to its mediator to broadcast the message through the cluster. Dashed arrows, which link all replicas, represent the communication established by the two secondary actors: listener and mediator.

[2]See `https://github.com/google/leveldb/`
[3]See `http://cassandra.apache.org/`
[4]See `https://cassandra.apache.org/doc/latest/cql/`

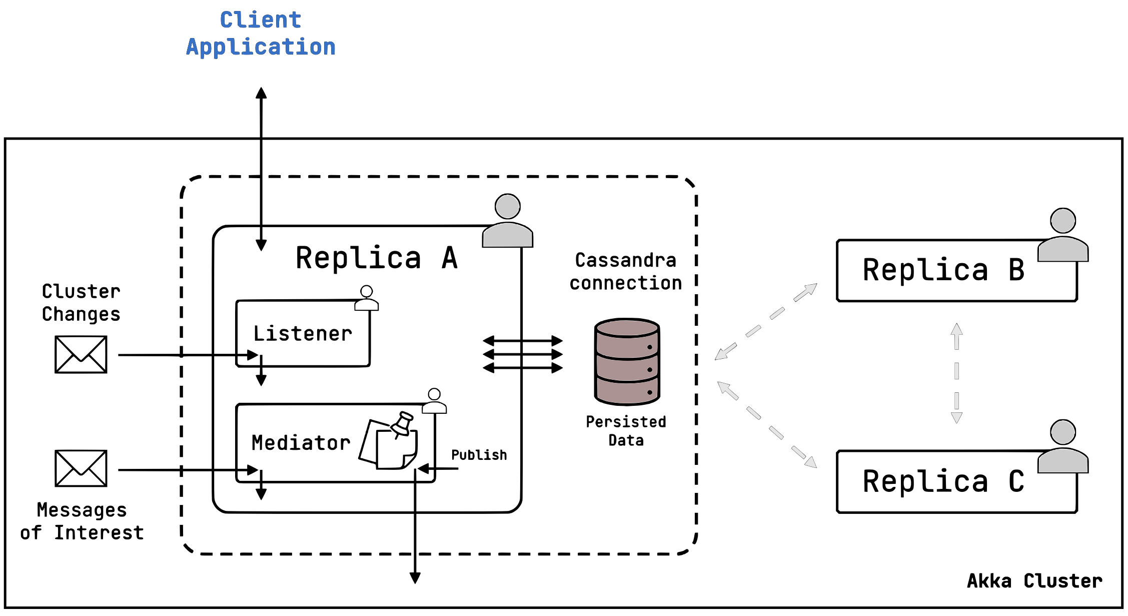


Figure 4.7: Architecture diagram.

Chapter 5

Dynamic Cluster

The following chapter complements the previous one. Therefore, it is assumed that the reader already has an understanding of the architecture that supports our approach.

We will address relevant aspects of the implementation, such as how the broadcast works, the stabilization of operations, and how we develop a dynamic cluster environment. The latter includes several steps, from data replication to the beginning of the inclusion of the new replica in stability verification.

5.1 Assumptions

Before moving into implementation details, as part of it, we assume the following:

- A reliable communication, with no message duplication, corruption, or loss of messages, and they are eventually delivered to the recipient;
- A replica does not fail, though it is acceptable to stop responding for a finite amount of time and eventually recover to an active state;
- Every replica is a publisher and a subscriber.

5.2 Message Passing

A replica contains the *mediator* actor who is responsible for broadcasting messages on the cluster. Internally, the transmission of messages is eventually consistent.

A part of our architecture contains the publish/subscribe pattern [17]. We developed the cluster using this pattern so that each replica does not need to know the location of the others on the network to exchange messages. Replicas publish messages concerning a topic, and those who have subscribed to that topic receive them. Our approach covers four topics: *states*, *ready*, *operations*, and *acks*. The topics *states* and *ready* relate to joining replicas, and we will address them in a subsequent section on this chapter. The topic *operations* refers to the operations originated in a replica, which we call a local execution. Then, they are broadcast, regarding this topic, so that the receivers can execute it as a remote operation. Note that the successful execution of a remote operation can only occur if the replica meets all operation's dependencies. Otherwise, it becomes a pending operation and awaits the arrival of each missing dependency. The topic *acks* is relevant when it comes to the stability of operations, which we will present in the following section.

5.3 Stability Check

The stability of an operation allows us to reduce the size of the replica's internal state. The reason is that stable operations do not need to be broadcasted since the whole cluster is aware of their existence. We delete the records of the operation in the data structures of the replica's internal state (`executedOps` and `opNamesMapping`), using the `origin` field. Also, we add a new record to the persistent stability table, which stores the information about stable operations.

As soon as a replica executes an operation, local or remote, it publishes a message related to the topic *acks*, which holds the `origin` of the operation. Remember that the field `origin` uniquely identifies the operation, and holds the replica identifier and counter that the replica had at the time of creating the operation.

Algorithm 2 Stability Check

```
1: function RECEIVE_ACK(origin)        ▷ origin field has type (replica_id, lamport_counter)
2:     op ← executedOps[origin]

3:     if op ≠ null then
4:         opUpdated ← op.copy(delivered++)
5:         executedOps ← executedOps ∪ {⟨origin, opUpdated⟩}
6:         CHECK_STABILITY(origin)
7:     else
8:         updatePendingAcks(origin)

9: function CHECK_STABILITY(origin)
10:    op ← executedOps[origin]

11:    if op.delivered == op.lastAckId - 1 then                 ▷ -1 to exclude itself
12:        updateInternalState(origin)      ▷ delete references of origin from data structures
13:        stabilize(persistenceId, op, origin)
```

As already mentioned in Section 4.2.1, each replica knows the size of the cluster. The delivery of an ack implies the verification of the stability regarding an origin, as Algorithm 2 presents. An operation, extending what we have already mentioned in Section 4.1, contains the following: (i) an object `Origin`, which has the id of the replica and the counter at the moment of its creation, (ii) a name, (iii) a list of parameters of any type, (iv) a set of dependencies, and (iv) a field called `delivered`, which is an integer that represents the number of acks that the replica holds related to this operation.

When the number of acks reaches the size of the cluster, the operation becomes stable since all nodes know the operation and have it in their state ($L11-13$). In case the replica receives an ack referring to an unknown operation, then the ack turns pending the arrival of that operation. The reason is that this ack will be used to stabilize the operation when it arrives. Otherwise, if we did not keep this information, we would have to ask if every replica had the operation. As soon as the operation arrives, the field `delivered` is incremented according to the number of pending acks that the operation has ($L8$).

This approach works well in the case of a static cluster environment. The challenge is when we consider a dynamic cluster. Imagine that initially, we have three replicas. All of them wait for the arrival of two acks for stabilization. In case a new replica joins when the others are exchanging acks, some replicas may stabilize the operation, while others do not. The reason is that they may have late knowledge that the size has increased. To overcome such a problem, we decided to add a new field to the definition of the operation. The field is an integer that matches the size of the cluster at the moment of the operation's creation. We assign replicas identifiers in an orderly manner, so having three replicas assumes that the replica with the third identifier will be the last one included to participate in the stability. Therefore, even with a dynamic environment, new replicas are not considered in the field delivered of previously created operations. So one might ask: "*How the new entry stabilizes this operation?*". Well, the new replica may or may not receive the operation and all acks. We will address this issue in the next section.

5.4 Dynamic Cluster Environment

Developing a dynamic environment of replicas is difficult, and the challenge is to maintain an initial up-to-date state of the new replica without loss of messages. The approach that we will present considers a **join model**, in which replicas do not require starting from the same initial state to have a new node in the cluster. In other words, no synchronization is required between the existing replicas, as they keep their state in receiving mode. We consider that only one replica works as a join node, which we call the *replicator*.

The *replicator* is the node responsible for replicating the data upon join requests from new nodes. This data includes the internal state of the replica, as well as the persisted data in the database. If we only replicate the persisted data, then state convergence would not be guaranteed, as certain operations could be lost.

5.4.1 Two-Phase Joining

Initially, a new instance of a replica, which is a persistent actor, enters a recovery mode, followed by a receiving mode which starts accepting messages to process. The replica also creates the instance of the *mediator*, i.e., the actor responsible for the retrieval and publication of messages. The *mediator* of a new replica N subscribes to the topics *states* and *ready*. The **topic *states*** refers to the transmission and acquisition of internal states. The new entries and the *replicator* are the replicas that subscribe to this topic. While the *replicator* subscribes to the topic permanently, N unsubscribes upon receiving a state from the *replicator*. The **topic *ready*** informs the cluster that N is ready to participate. As soon as N applies the received state, it broadcasts a "ready message" and can start receiving operations and acks. As such, the stability calculation for instances of operations originated from the moment that the cluster acknowledges N will include N.

The joining process is divided into two phases, where the purpose of the second phase is to ensure state convergence in case the first phase is not sufficient. Following the illustration in Figure 5.1, let's assume that a replica with id r4 intends to join the cluster and, for simplicity, that the *replicator* has id r1. The actor shown in the center of the figure is the *mediator*. In practice, there are two *mediator* actors, one for each replica. For simplicity, we are only presenting a single *mediator* as the channel of communication. We consider that the cluster has three replicas before the join request from r4. For an easier understanding, we show only the *replicator* and the new replica. When necessary, we will specify the case of exchanging messages with the other existing replicas. The arrows represent the exchange of messages.

1st Phase

Initially, r4 sends a state request to r1. r1 replicates its persistent data to r4. Eventually, r1 sends its internal state S1 to r4. The replica r1 also stores the state for future reference. Note that in the courseware application, replicating the persistent data means that we will have duplicated data from student, course, enrollment, and stability tables. As we are using the Cassandra database, being it distributed, it has several nodes. The nodes are arranged according to the primary key, which is called the partition key. Data that has the same partition key, without considering intervals of identifiers, are in the same node. All tables have the replica identifier as the partition key. Thus, r4 will have the data persisted with its identifier.

Next, r4 applies S1 and subscribes to the **topics *operations*** and ***acks***. Then, r4 calculates the minimum counter of S1, which is obtained using the field `origin` of the operations. Finally, r4 broadcasts the message indicating that it is ready to join the cluster. Afterward, all replicas increase the size of the cluster. Each replica replies to r4 by broadcasting a message. This message has the id of the receiver (r4) and the value of `counterOps`, which belongs to the internal state. As soon as r4 receives all replies, it calculates the maximum counter. r4 informs r1 of the minimum and maximum counter that it acknowledges, and r1 replicates all operations that pertain to the interval $[min, max]$ and that have not yet been replicated. These operations expect three acks to stabilize since we are considering the existence of three replicas in the cluster. Finally, r1 tells r4 that the process ended, attaching to the message a boolean referencing if it has replicated

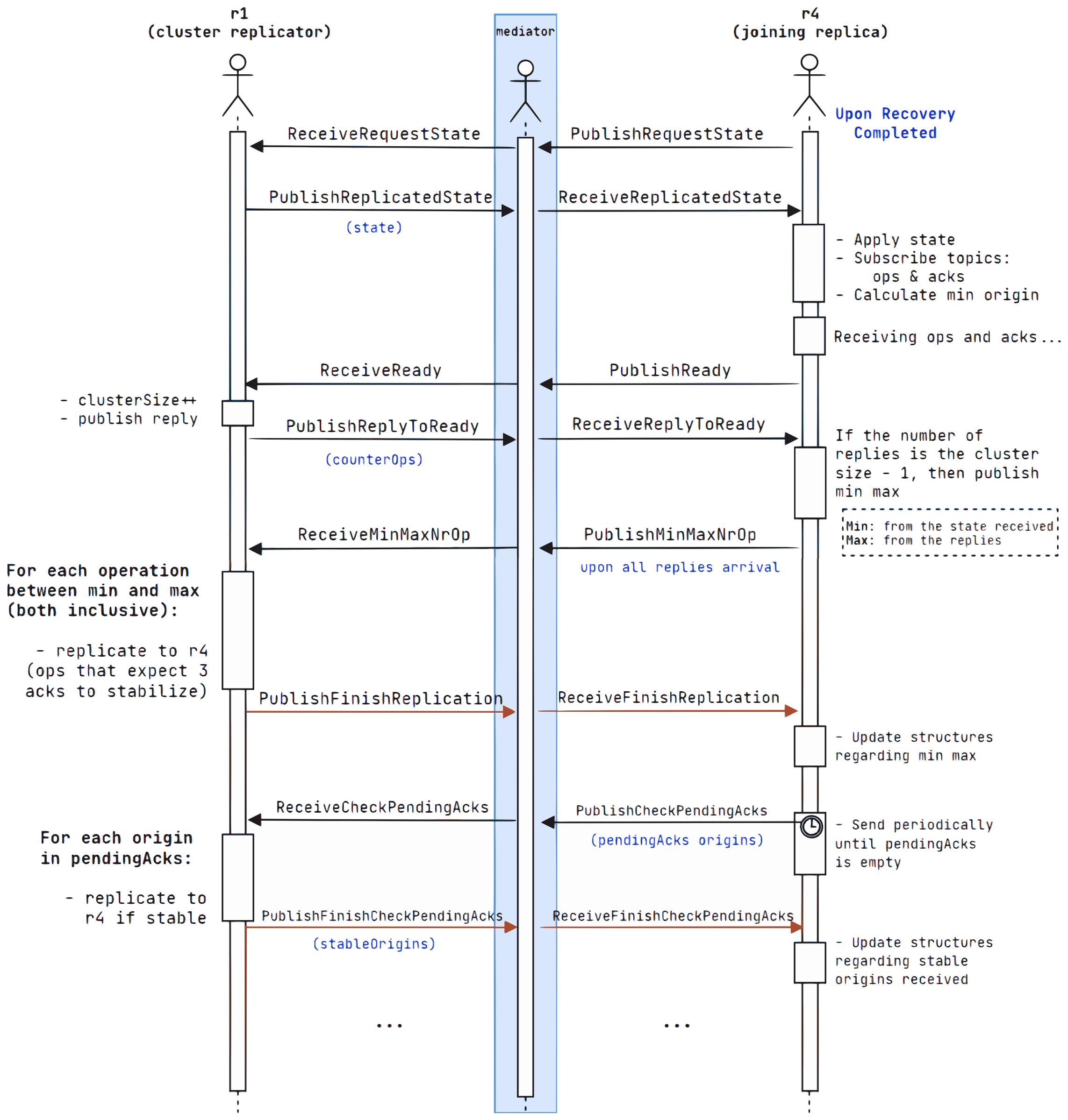

r1
(cluster replicator)
mediator
r4
(joining replica)
Upon Recovery Completed
ReceiveRequestState
PublishRequestState
PublishReplicatedState
(state)
ReceiveReplicatedState
- Apply state
- Subscribe topics: ops & acks
- Calculate min origin
Receiving ops and acks...
ReceiveReady
PublishReady
- clusterSize++
- publish reply
PublishReplyToReady
(counterOps)
ReceiveReplyToReady
If the number of replies is the cluster size - 1, then publish min max
Min: from the state received
Max: from the replies
ReceiveMinMaxNrOp
PublishMinMaxNrOp
upon all replies arrival
For each operation between min and max (both inclusive):
- replicate to r4 (ops that expect 3 acks to stabilize)
PublishFinishReplication
ReceiveFinishReplication
- Update structures regarding min max
ReceiveCheckPendingAcks
PublishCheckPendingAcks
(pendingAcks origins)
- Send periodically until pendingAcks is empty
For each origin in pendingAcks:
- replicate to r4 if stable
PublishFinishCheckPendingAcks
(stableOrigins)
ReceiveFinishCheckPendingAcks
- Update structures regarding stable origins received
...
...

Figure 5.1: Two-phase joining diagram.

all the operations on the interval $[min, max]$. In case they haven't all succeeded, it attaches their `origin`. r4 updates its internal state according to what was received.

2nd Phase

During the first phase, r4 may receive acks from unknown operations. These acks remain pending until the corresponding operation arrives so they can be added to the field acks of that operation. Well, these operations may never arrive because they have become stable. For r4 to recognize this stabilization, it will have to ask r1. To do so, r4 sends a request to r1 to verify each pending ack, attaching in the message the field `origin`. Then, r1 will query the stability table for a record of each `origin`. If it finds, it replicates with the persistence id of r4. Then it replies with the set of `origin` it has stabilized. Finally, after receiving the response, r4 has to update its internal state accordingly.

This phase repeats periodically as much as necessary until r4 has the set of pending acks empty. Hence, if we consider that before the introduction of r4, we had three replicas, all operations whose stability awaits three acks will be made stable for r4. Note that this process is required, as there is the case of the new replica not recognizing certain operations while preparing its join in the first phase.

5.4.1.1 Optimization

After several runs and experimenting with the 2-phase joining algorithm, we noticed that there was the possibility to expedite the process of state convergence. In particular, by decreasing the overhead in the replication of operations that become stable before the new replica indicates its preparation.

In Figure 5.2 we highlight the improvements concerning what was previously presented in Figure 5.1. As soon as r1 (the *replicator*) sends its state to the new replica (r4), r1 will add a new entry in a map, which we call persistedOpsLater, a key-value pair, whose key is the id of the new replica and the value is a set of Origin objects. Replica r1 stores all the operations that become stable until the new replica signals that it is ready. When r1 receives the preparation message from r4, it replicates all the operations corresponding to the field origin in persistedOpsLater. Replica r1 informs r4 that it has terminated the process, and r4 updates its status accordingly. Then, r1 on receiving the message with the minimum and maximum counter filters those that have not yet replicated. Afterward, when the second phase is completed, that is, upon state convergence, r1 updates the map by removing the entry corresponding to the identifier of the new replica.

Hence, we anticipate the potential overhead in the database when replicating all operations that are in the interval $[min, max]$, i.e., the minimum and maximum counter that the new replica acknowledges upon receiving all replies to its preparation.

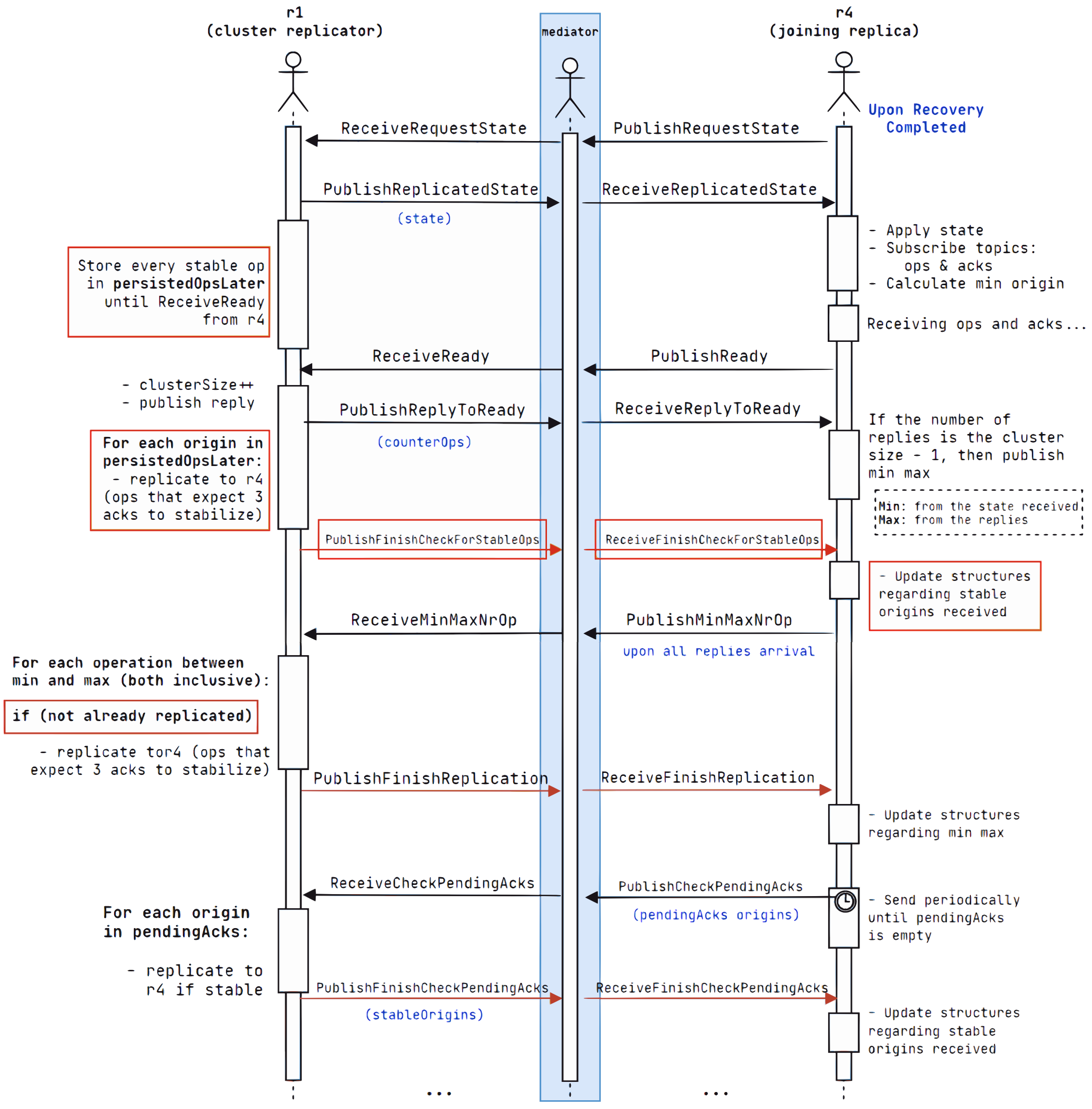

r1
(cluster replicator)
mediator
r4
(joining replica)
Upon Recovery Completed
ReceiveRequestState
PublishRequestState
PublishReplicatedState
ReceiveReplicatedState
(state)
- Apply state
- Subscribe topics: ops & acks
- Calculate min origin
Store every stable op in persistedOpsLater until ReceiveReady from r4
Receiving ops and acks...
ReceiveReady
PublishReady
- clusterSize++
- publish reply
PublishReplyToReady
ReceiveReplyToReady
(counterOps)
If the number of replies is the cluster size - 1, then publish min max
For each origin in persistedOpsLater:
- replicate to r4 (ops that expect 3 acks to stabilize)
Min: from the state received
Max: from the replies
PublishFinishCheckForStableOps
ReceiveFinishCheckForStableOps
- Update structures regarding stable origins received
ReceiveMinMaxNrOp
PublishMinMaxNrOp
upon all replies arrival
For each operation between min and max (both inclusive):
if (not already replicated)
- replicate tor4 (ops that expect 3 acks to stabilize)
PublishFinishReplication
ReceiveFinishReplication
- Update structures regarding min max
ReceiveCheckPendingAcks
PublishCheckPendingAcks
(pendingAcks origins)
- Send periodically until pendingAcks is empty
For each origin in pendingAcks:
- replicate to r4 if stable
PublishFinishCheckPendingAcks
ReceiveFinishCheckPendingAcks
(stableOrigins)
- Update structures regarding stable origins received
...
...

Figure 5.2: Two-phase joining diagram with optimization.

Chapter 6

Evaluation

In this chapter, we present the experimental evaluation of our algorithm's performance. Overall, our primary goal is to have our algorithm's performance located amidst the two weaker consistency models in the spectrum, presented in Section 2.4: eventual consistency and causal consistency. The evaluation focuses on comparing execution times concerning consistency models, varying the number of operations performed in the cluster replicas.

6.1 Configuration

We evaluated the performance over two case studies: the courseware (described in Section 2.6.1) and a synthetic application. We developed the latter application to assess and compare the performance according to an application that had an operation with a significant number of *parameter dependencies*, which in this case contains five. Figure 6.1 illustrates the dependency graph of the synthetic application. Note that the graph construction follows the description presented in Section 4.1.1.

The way we implemented the system allows us to instantiate a replica specifying

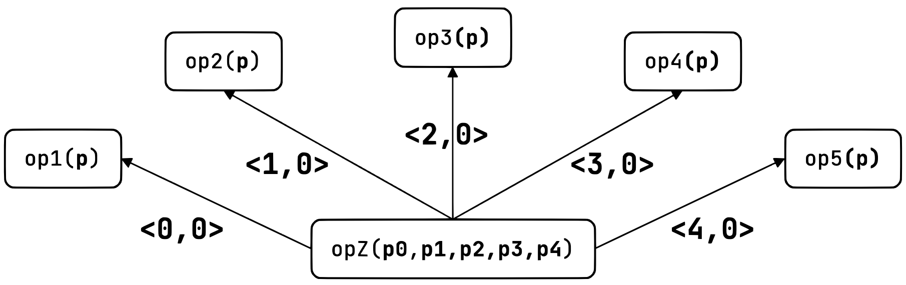


Figure 6.1: Synthetic dependency graph.

whether we want to look at the parameters or break the application invariant, as described in Table 6.1. Essentially, the key lies in how to build dependencies. In the case of eventual consistency, it performs write operations in any order, without restriction. Therefore, the application invariant can break. On the other hand, causal consistency requires a partial order, in which all operations preceding an operation `A` have to be observed by others before `A`. Our implementation of causal consistency is optimized, in the sense that the set of dependencies of `A` will have all non-stable operations at the moment of its execution, instead of the set of all operations that happened before. This results in less dependencies compared to a standard causal consistency implementation.

Table 6.1: Consistency model configuration.

Consistency Model	`lookAtParameters`	`letInvariantBreak`
Eventual Consistency (EC)	true or false	true
Semantic Consistency (SC)	true	false
Causal Consistency (CC)	false	false

As described in Section 4.2.1, each actor has a dispatcher, which allows it to schedule the processing of a message. Thus, we can introduce an interval regarding the transmission of an operation, more specifically what we call a broadcast delay. This delay consists of message passing between the replica actor and the mediator actor, who is responsible for the broadcast. In this manner, we decided to simulate different distances between clients. Table 6.2 presents three different latency intervals and associated locations, which rely on Amazon AWS latency monitoring. [1]

Table 6.2: Simulation of different distances between clients.

Location		**Latency Interval (ms)**
From	**To**	
Singapore	Tokyo	[50, 100]
São Paulo	Canada	[100, 150]
London	Sydney	[200, 300]

We generate (i) a set of random replicas, (ii) a set of random operations that includes the operation name and parameters, and (iii) a set of random values of broadcast delay, and register the execution details into a file. The reason is that for a given number of operations and a broadcast delay interval, we want the set of operations to be the same for each consistency model configuration. Each execution instruction of the file has the following format, represented as a regular expression:

$$replica\_id \quad operation\_name \quad (parameter(,parameter)^*)? \quad broadcast\_delay$$

[1] See https://www.cloudping.co/grid and https://aws.amazon.com/

The broadcast delay value is random within the chosen interval delimiters. For example, if we are using the courseware application and the range [100,150], an execution instruction of the file could be the following:

$$1 \quad enroll \quad s1, c1 \quad 122$$

To conduct a semantic analysis, requesting the execution of a certain number of operations involves avoiding throwing exceptions, particularly when it might lead to an invariant violation, such as the non-existence of a course or student when writing an `enroll`. In other words, the preconditions must be satisfied. To do so, we start by choosing a random replica, then the operations without dependencies, and, finally, the operation that has those operations as dependencies. For instance, requesting the execution of five operations, namely one `addCourse`, two `registerStudent`, and two `enroll` could result in the following execution instructions:

1	*addCourse*	$c1$	135
1	*registerStudent*	$s2$	117
1	*enroll*	$s2, c1$	122
2	*registerStudent*	$s1$	141
2	*enroll*	$s1, c1$	103

When reading the file, for each execution instruction, we send a request to the replica with the given identifier. The replica will process the local execution of the operation with the given name, parameters, and broadcast delay.

The experimental evaluation comprises an Akka cluster with three replicas in which all share the same configuration, running on a single machine with Windows 10 64bits version 1909, 8 GB memory, and Intel Core i5-8265U quad-core CPU @ 1.60 GHz with 8 hardware threads. The evaluation results in the comparison between consistency models, considering the average execution time required to perform a given number of operations. We perform different experiments for each latency interval presented in Table 6.2.

The results we will present include: (i) publication and retrieval of messages, (ii) communications with the database, (iii) construction of dependencies when executed locally, (iv) verification of dependencies in a remote operation, (v) potential triggers of pending operations, (vi) acks processing, and (vii) stabilization of operations. These results will be discussed in the following section.

6.2 Evaluation Results

For both courseware and synthetic applications, we present the results regarding the execution of 512, 1024, 2048, and 4096 operations. Tables 6.3 and 6.4 expose their distribution.

Table 6.3: Courseware application: distribution of operations.

addCourse	*registerStudent*	*enroll*	$\sum$
12	200	300	**512**
24	400	600	**1024**
48	800	1200	**2048**
96	1600	2400	**4096**

Table 6.4: Synthetic application: distribution of operations.

*op*1	*op*2	*op*3	*op*4	*op*5	*opZ*	$\sum$
56	58	68	62	68	200	**512**
112	116	136	124	136	400	**1024**
224	232	272	248	272	800	**2048**
448	464	544	496	544	1600	**4096**

Figures 6.2 and 6.3 illustrate the results obtained. The eventual consistency (EC) and semantic consistency (SC) reveal a more linear behavior than the causal consistency (CC). In the latter, the execution times are irregular.

The reason is that in eventual consistency, there is no effort in ensuring that preconditions are satisfied, however, invariants might be broken. On the other hand, semantic consistency is more precise in the sense that it always has the guarantee that the number of dependencies has a fixed upper-bound. Consider the different case scenarios of *addCourse* and *registerStudent* to stabilize when executing an *enroll* that depends on them semantically. In case neither of the dependencies is stable, then *enroll* has two dependencies. Otherwise, it has zero or one, if both are stable or one is stable, respectively. With causal consistency, it has as many dependencies as non-stable operations. Therefore, when we request the execution of a high number of operations, the message queue of the replicas will increase rapidly, and it will take longer to stabilize, leading to an increase in the set of dependencies of each operation.

Occasionally, having a significant or minor execution time difference between causal consistency and semantic consistency is perfectly acceptable. The reason is that we are dependent on the following factors: (i) the set of operations written in the file, (ii) the order in which they are executed and received, and (iii) the ratio in which operations stabilize, which will affect the number of dependencies. These factors are evident from the results obtained in Figure 6.2 for 2048 operations and Figure 6.3 for 1024 operations, where the execution time difference between SC and CC is not significant.

Typically we expect an increase in the difference between CC and SC as the number of operations increases. We start to verify this increase from around 4096 operations.

We notice that in an application with fewer dependencies, the difference progresses gradually. Yet, in synthetic, there are sudden changes, which are also more irregular. In EC, as it only executes, the growth of dependencies makes no impact. In the case of SC, it increases by approximately 14.13%, as presented in Table 6.6. As for CC, the changes are more unstable, in particular, when increasing the number of operations.

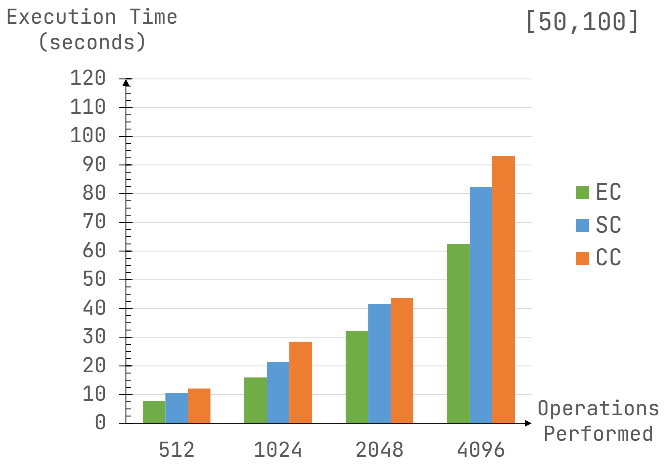
Our approach remains approximately constant, even as the broadcast interval increases. As already mentioned, this is because we have a fixed maximum number of dependencies, and the rate at which they stabilize is not very distinct. The same happens with EC. However, this is not the case with CC since the ratio of operations that need to stabilize is higher and takes longer. In particular, using the broadcast interval between 200 and 300 ms, where the number of pending operations increases rapidly.

Additionally, we present in Table 6.5 a linear regression model to estimate values for higher numbers of operations. Although the values of CC are not properly linear, we have also decided to present a linear estimate to maintain the same comparison between the models. The “Linear Fit” column corresponds to the linear equation $y = mx + b$ derived from the values in the “Actual Value” column. The latter corresponds to the values extracted from experimentation, presented in Figure 6.3 (b). The next column contains the difference between the current value and the equation. We can see that the values for EC and SC differ by less than a second, but for CC, as expected, this does not happen.

To reduce the difference and obtain a value closer to the real prediction, we carry out another linear trendline equation for the remainder or residual value. The last column presents the predicted values. We can verify that the difference starts to be less significant from 4096 operations. The results for a larger number of operations serve as a reference.

Overall, the estimation confirms the assumption that the differences between SC and CC become more significant as the number of operations increases.

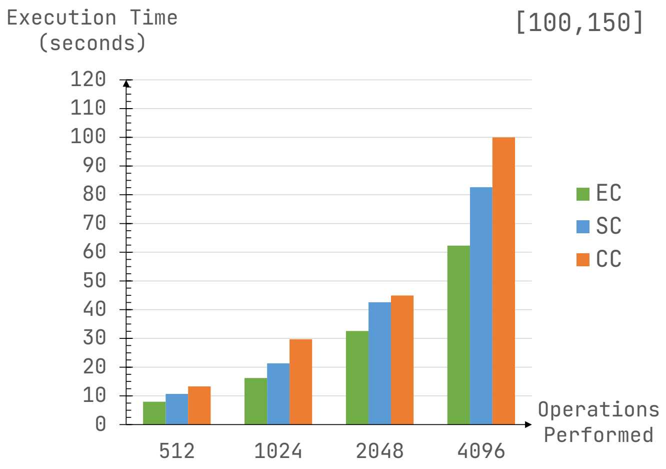
Our solution achieves the same guarantees as causal consistency, namely invariants preservation. However, we propose a refined approach in the sense that pending operations genuinely await those on which they are dependent semantically. Semantic analysis confirms that we can improve in terms of performance, ensuring application correctness.



(a)

[50,100]			
	EC	SC	CC
512	7.8280	10.5800	12.1120
1024	15.9807	21.3130	28.4450
2048	32.1315	41.5155	43.6735
4096	62.5055	82.3532	93.0925

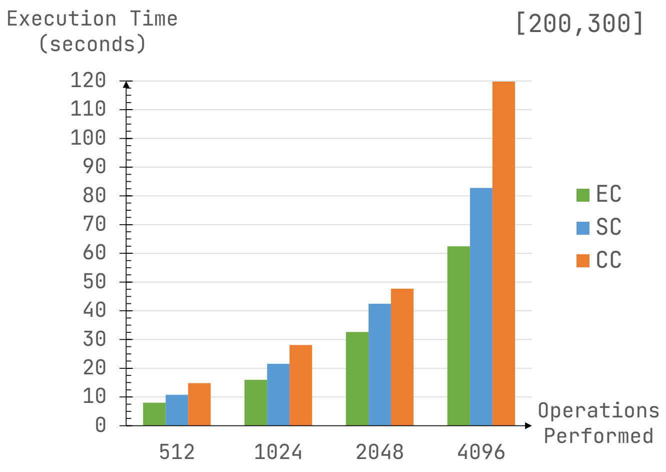
(b)



(c)

[100,150]			
	EC	SC	CC
512	7.9487	10.6977	13.2907
1024	16.2325	21.3133	29.7193
2048	32.5760	42.5770	44.9277
4096	62.3135	82.6310	99.9795

(d)

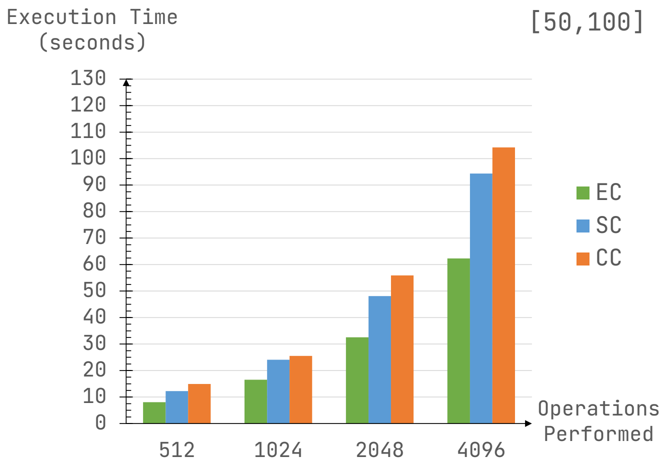


(e)

[200,300]			
	EC	SC	CC
512	7.9930	10.7875	14.8540
1024	15.9813	21.5520	28.1153
2048	32.6500	42.4503	47.6993
4096	62.4393	82.7867	119.7740

(f)

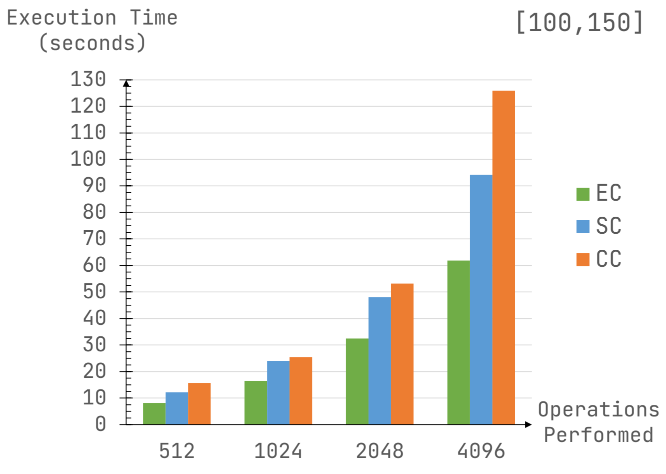
Figure 6.2: Results of the courseware application, considering different latency intervals.



(a)

[50,100]			
	EC	SC	CC
512	8.0503	12.2278	14.9158
1024	16.5223	24.0604	25.5046
2048	32.5417	48.1342	55.8976
4096	62.3063	94.3785	104.2205

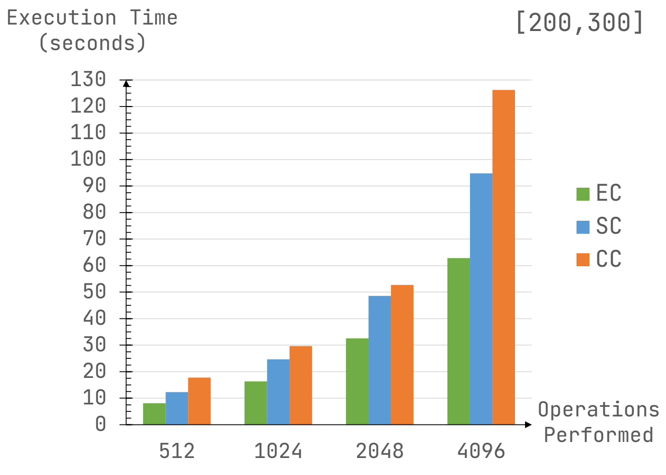
(b)



(c)

[100,150]			
	EC	SC	CC
512	8.1780	12.1664	15.7270
1024	16.4923	24.0226	25.4542
2048	32.4650	48.0833	53.1555
4096	61.8663	94.2335	125.8705

(d)



(e)

[200,300]			
	EC	SC	CC
512	8.0920	12.2843	17.7828
1024	16.3490	24.6527	29.6555
2048	32.5300	48.5790	52.7178
4096	62.8340	94.7773	126.2705

(f)

Figure 6.3: Results of the synthetic application, considering different latency intervals.

Table 6.5: Results estimation of the synthetic application, relying on a linear trendline equation.

The first column refers to the given number of operations performed.
The results refer to a broadcast delay in the interval **[50,100] ms**.
All values are represented in **seconds**.

Linear Fit Equation:
(EC) $y = 0.0152075203x + 0.5253520833$
(SC) $y = 0.0230229834x + 0.3968775000$
(CC) $y = 0.0254718701x + 0.9829075000$

Residual Fit Equation:
(EC) $y = 0.0000436392x + 0.4512741734$
(SC) $y = 0.0000402300x + 0.2251647478$
(CC) $y = 0.0000943078x + 1.3109821400$

	Actual Value	**Linear Fit**	**Residual Value** |Linear Fit - Actual Value|	**Residual Fit**	**Predicted Value** Linear Fit - Residual Fit
			EC		
0	**0.0000**	0.5254	0.5254	0.4513	**0.0741**
512	**8.0503**	8.3116	0.2613	0.4736	**7.8380**
1024	**16.5223**	16.0979	0.4244	0.4960	**15.6019**
2048	**32.5417**	31.6704	0.8713	0.5406	**31.1297**
4096	**62.3063**	62.8154	0.5091	0.6300	**62.1853**
8192	-	125.1054	-	0.8088	**124.2966**
16384	-	249.6854	-	1.1663	**248.5191**
32768	-	498.8454	-	1.8812	**496.9641**
65536	-	997.1654	-	3.3112	**993.8542**
			SC		
0	**0.0000**	0.3969	0.3969	0.2252	**0.1717**
512	**12.2278**	12.1846	0.0432	0.2458	**11.9389**
1024	**24.0604**	23.9724	0.0880	0.2664	**23.7061**
2048	**48.1342**	47.5479	0.5863	0.3076	**47.2404**
4096	**94.3785**	94.6990	0.3205	0.3899	**94.3091**
8192	-	189.0012	-	0.5547	**188.4464**
16384	-	377.6054	-	0.8843	**376.7211**
32768	-	754.8140	-	1.5434	**753.2706**
65536	-	1509.2311	-	2.8617	**1506.3694**
			CC		
0	**0.0000**	0.9829	0.9829	1.3110	**0.3281**
512	**14.9158**	14.0245	0.8913	1.3593	**12.6652**
1024	**25.5046**	27.0661	1.5615	1.4076	**25.6585**
2048	**55.8976**	53.1493	2.7483	1.5041	**51.6452**
4096	**104.2205**	105.3157	1.0952	1.6973	**103.6184**
8192	-	209.6485	-	2.0836	**207.5649**
16384	-	418.3140	-	2.8561	**415.4579**
32768	-	835.6451	-	4.4013	**831.2439**
65536	-	1670.3074	-	7.4915	**1662.8158**

Table 6.6: Growth rate of semantic consistency from an application with two to five dependencies.

The values refer to those shown in Figures 6.2 and 6.3 for semantic consistency (SC). That is, the time it takes to execute a given number of operations using a certain broadcast interval. The first column refers to the given number of operations performed. All values are represented in **seconds**.

	SC Rate			
	[50,100] ms (s)	**[100,150] ms (s)**	**[200,300] ms (s)**	**Mean (s)**
	Courseware			
512	10.58	10.70	10.79	**10.69**
1024	21.31	21.31	21.55	**21.39**
2048	41.52	42.58	42.45	**42.18**
4096	82.35	82.63	82.79	**82.59**
	Synthetic			
512	12.23	12.17	12.28	**12.23**
1024	24.06	24.02	24.65	**24.24**
2048	48.13	48.08	48.58	**48.26**
4096	94.38	94.23	94.78	**94.46**

The table below presents the growth ratio from the courseware application that has two dependencies to the synthetic application that has five dependencies. The mean values that were used are presented in the table above.

	Growth Rate	
	Percentage $\frac{MeanSynthetic - MeanCourseware}{MeanCourseware} * 100$	**Mean Percentage**
512	14.41 %	**14.13 %**
1024	13.32 %	
2048	14.41 %	
4096	14.37 %	

Chapter 7

Conclusion

Large scale distributed applications are used to serve millions of users and, to fulfill their needs, it is necessary to ensure availability and to have a consistent system. However, according to the CAP theorem [19], a partitioned distributed system cannot fully guarantee both availability and consistency. If one places the consistency models in a spectrum, on one edge, one finds the weak consistency models, such as the eventual consistency. This consistency model guarantees high availability without maintaining any order between operations. Another weak consistency model is causal consistency, where although it has tighter network requirements compared to eventual consistency, it guarantees data integrity through partial order between operations, even during network partitions. On the other edge, one finds the strong consistency models, such as sequential consistency, in which replicas behave as a singleton, ensuring consistency.

Our initial goal was to develop an approach that stood between eventual consistency and causal consistency, i.e., to enable high availability of eventual consistency while ensuring the integrity of the data, as is the case with causal consistency. Generally, approaches that use causal consistency make use of vector clocks, which are designed to determine the partial order between events. However, the main drawback is that it requires every replica to maintain a vector of the size of the cluster, which grows linearly according to the number of existing replicas. However, for a high number of replicas, broadcasting the vector clock may become unfeasible. Even with the existing alternatives and improvements regarding the limitation of the data sent over the network, as mentioned in Chapter 3, generally in the internal state, it is necessary to maintain a vector of a size corresponding to the number of replicas.

In this dissertation, we propose an approach that uses a consistency model called semantic consistency (SC), in which we provide better performance compared to causal consistency, while still ensuring data integrity. To do so, we rely on CISE3 [42], a tool for static analysis of weakly consistent applications, to manage a semantic analysis. The static analysis of the client application provides dependencies between operations, including the relationship between their parameters. As a result, we build a graph of dependencies, in which the nodes are the operations, and the edges are the relationships between the parameters. The algorithm we developed uses this graph to build the dependencies. Thus, for instance, the enrollment of a student `s1` in a course `c1` is not dependent on all the student and course registrations that the replica observed before executing the enrollment. It is only dependent on two operations, being the registration of `s1` and `c1`. Through our graph construction, we ensure that the number of maximum dependencies is always settled for a given operation throughout the program life cycle.

We also present the architecture that supports our approach, which is based on Akka [29]: an actor-based middleware that provides a set of open-source libraries that allows the development of distributed and concurrent applications. The replicas, which are remote actors, form a cluster that supports the architectural pattern publish/subscribe [17]. The purpose of using this pattern was to abstract direct communication between the replicas, that is, to avoid them communicating directly with each other, not knowing the exact location of the other members in the network to exchange messages. Moreover, the architecture also includes persistence. We use the database NoSQL Apache Cassandra [1], which is widely used in distributed environments since it guarantees high availability. For each case study, we develop the corresponding data model. We always include a table that stores stable operations: a stability table. An operation is considered stable when it is observed by all replicas. In this sense, the algorithm we have developed performs the verification and manages stability. Whenever an operation becomes stable, it is removed from the internal state of the replica and added a new record to the stability table.

Furthermore, we specified how we handled the integration of a dynamic cluster environment. One of the replicas, typically the oldest, assumes the role of the replicator. The replicator is responsible for receiving new join requests to the cluster and handles the replication of data for the new replica. Through a two-phase joining, we guarantee state convergence of the new replica, as well as its integration into the stability of operations.

Lastly, we evaluate the performance of the developed algorithm, taking into account the number of operations performed, a latency interval, which simulates several realistic distances, as well as two case studies. We confirm that as the overhead in the number of executed operations increases, we start to have significant improvements in terms of performance, compared to the use of causal consistency.

Summary

To summarize, we were able to present a solution that reduces the synchronization between the replicas while ensuring data integrity, which was the primary goal of the

[1] See `http://cassandra.apache.org/`

dissertation.

In an early stage, one of the concerns focused on identifying a technique to avoid the use of vector clocks to track causality. The way we built the graph of dependencies allowed us to establish a fine-grained analysis of dependencies. We managed to reduce the partial order between operations that the causal consistency maintains due to the semantic analysis and the consequently limited number of dependencies. For instance, consider two replicas `r1` and `r2`. In the case of `r1` sends an operation `A` to `r2`, `r2` is not required to observe all the updates that occurred before `A`, to apply `A`.

The usage of Akka helped us to avoid writing low-level code, more precisely threads, and the concurrent accesses between them. It allowed us to focus on the development of the algorithm and the exchange of messages in the cluster.

We had some difficulties in developing the dynamic cluster environment. The reason is that the aim was to allow new replicas in the cluster without the existing ones having to synchronize and stop their execution. From what we searched, there is no such study in the literature. The study that comes closest to what we have developed involves synchronization, and all replicas have to recognize that a given replica is joining the cluster at a certain point [11]. So, in this context, state convergence became a challenge, and we managed to overcome it through the development of the two-phase joining.

The evaluation confirms that, in scenarios where we want to ensure availability and application correctness, the semantic analysis of operations provides performance gains when compared with a classic approach of causal consistency. It limits the number of dependencies and reduces the metadata that is broadcast on the network.

7.1 Contributions

In summary, this dissertation makes the following contributions:

1. A detailed explanation of the semantic consistency (SC) model, which provides availability, efficient updates convergence, and ensures application correctness by preserving the invariants.

2. The introduction of a semantic analysis of operations, as well as the proposed construction of the graph of dependencies.

3. The presentation and explanation of the architecture that supports our approach, built using the Akka actor-based middleware and the use of persistence.

4. The implementation of the protocol, which takes as input a set of dependencies between operations and the relationship between their parameters. This input is provided by the static analysis tool CISE3 [42].

5. The stability of operations using SC, establishing the relationship between the internal state of the replicas and the data that is persisted.

6. The detailed explanation and implementation of a cluster in a dynamic environment, divided in two phases, which allows new replicas to join the cluster without interrupting the execution of existing replicas.

7. An experimental evaluation that reveals the performance gains, which includes the following factors: (i) latency, (ii) number of operations performed, and (iii) consistency model. The evaluation is conducted using two case studies, which vary in the number of dependencies, and we measure the growth ratio in terms of execution time between them.

7.2 Future Work

To enhance our work, it would be valuable to formally prove the definition of the algorithm, to develop a specification that would confirm the correction properties, and to prove that the replicas converge to the same state in a geo-replicated setting using the protocol. We consider that it would be interesting to implement the specification in TLA+ [25] using the TLC model checker to validate the protocol.

The formal proof of the definition and convergence may be done using the logs that are generated when running the system. Our idea focuses on log analysis. We could formally confirm the construction of the dependencies and the stability of the operations by tracking the field `origin` of each operation. It is expected that in a distributed setting, the execution of the system keeps up and running, so this would have to be done in runtime. Additionally, it would also be possible to prove the state convergence of new replicas upon entering the cluster.

Bibliography

[1] A. Agarwal and V. K. Garg. “Efficient Dependency Tracking for Relevant Events in Shared-Memory Systems.” In: *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing*. PODC ’05. Las Vegas, NV, USA: Association for Computing Machinery, 2005, 19–28. ISBN: 1581139942. DOI: 10.1145/1073814.1073818. URL: https://doi.org/10.1145/1073814.1073818.

[2] P. Bailis. *Linearizability versus Serializability*. Accessed January 2020. 2014. URL: http://www.bailis.org/blog/linearizability-versus-serializability/.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. “Coordination Avoidance in Database Systems.” In: *Proc. VLDB Endow.* 8.3 (Nov. 2014), 185–196. ISSN: 2150-8097. DOI: 10.14778/2735508.2735509. URL: https://doi.org/10.14778/2735508.2735509.

[4] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. “Bolt-on Causal Consistency.” In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’13. New York, New York, USA: Association for Computing Machinery, 2013, 761–772. ISBN: 9781450320375. DOI: 10.1145/2463676.2465279. URL: https://doi.org/10.1145/2463676.2465279.

[5] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, and N. Preguiça. “IPA: Invariant-Preserving Applications for Weakly Consistent Replicated Databases.” In: *Proc. VLDB Endow.* 12.4 (Dec. 2018), 404–418. ISSN: 2150-8097. DOI: 10.14778/3297753.3297760. URL: https://doi.org/10.14778/3297753.3297760.

[6] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. “Putting Consistency Back into Eventual Consistency.” In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450332385. DOI: 10.1145/2741948.2741972. URL: https://doi.org/10.1145/2741948.2741972.

[7] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. “Towards Fast Invariant Preservation in Geo-Replicated Systems.” In: *SIGOPS Oper. Syst. Rev.* 49.1 (Jan. 2015), 121–125. ISSN: 0163-5980. DOI: 10.1145/2723872.2723889. URL: https://doi.org/10.1145/2723872.2723889.

[8] V. Balegas, C. Li, M. Najafzadeh, D. Porto, A. Clement, S. Duarte, C. Ferreira, J. Gehrke, J. Leitão, N. Preguiça, R. Rodrigues, M. Shapiro, and V. Vafeiadis. "Geo-Replication: Fast If Possible, Consistent If Necessary." In: *Bulletin of the Technical Committee on Data Engineering*. IEEE Data Engineering Bulletin, Special Issue on Data Consistency across Research Communities 39.1 (Mar. 2016), p. 12. URL: https://hal.inria.fr/hal-01350652.

[9] C. Baquero, P. S. Almeida, and A. Shoker. "Pure Operation-Based Replicated Data Types." In: *CoRR* abs/1710.04469 (2017). arXiv: 1710.04469. URL: http://arxiv.org/abs/1710.04469.

[10] C. Baquero and N. Preguiça. "Why Logical Clocks Are Easy." In: *Commun. ACM* 59.4 (Mar. 2016), 43–47. ISSN: 0001-0782. DOI: 10.1145/2890782. URL: https://doi.org/10.1145/2890782.

[11] J. Bauwens and E. Gonzalez Boix. "Memory Efficient CRDTs in Dynamic Environments." In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2019. Athens, Greece: Association for Computing Machinery, 2019, 48–57. ISBN: 9781450369879. DOI: 10.1145/3358504.3361231. URL: https://doi.org/10.1145/3358504.3361231.

[12] P. A. Bernstein and N. Goodman. "Concurrency Control in Distributed Database Systems." In: *ACM Comput. Surv.* 13.2 (June 1981), 185–221. ISSN: 0360-0300. DOI: 10.1145/356842.356846. URL: https://doi.org/10.1145/356842.356846.

[13] I. S. E. I. Bertrand Meyer and Usa. "Design by Contract: Making Object-Oriented Programs That Work." In: *Proceedings of the Technology of Object-Oriented Languages and Systems - Tools-25*. TOOLS '97. USA: IEEE Computer Society, 1997, p. 360. ISBN: 0818684852.

[14] K. P. Birman and T. A. Joseph. "Reliable Communication in the Presence of Failures." In: *ACM Trans. Comput. Syst.* 5.1 (Jan. 1987), 47–76. ISSN: 0734-2071. DOI: 10.1145/7351.7478. URL: https://doi.org/10.1145/7351.7478.

[15] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev. "Serializability for Eventual Consistency: Criterion, Analysis, and Applications." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: Association for Computing Machinery, 2017, 458–472. ISBN: 9781450346603. DOI: 10.1145/3009837.3009895. URL: https://doi.org/10.1145/3009837.3009895.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-Value Store." In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: https://doi.org/10.1145/1323293.1294281.

[17] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. “The Many Faces of Publish/Subscribe.” In: *ACM Comput. Surv.* 35.2 (June 2003), 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078. URL: https://doi.org/10.1145/857076.857078.

[18] C. J. Fidge. “Timestamps in message-passing systems that preserve the partial ordering.” In: *Proceedings of the 11th Australian Computer Science Conference* 10.1 (1988), 56–66. URL: http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf.

[19] S. Gilbert and N. Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services.” In: *SIGACT News* 33.2 (June 2002), 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

[20] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. “’Cause I’m Strong Enough: Reasoning about Consistency Choices in Distributed Systems.” In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, 371–384. ISBN: 9781450335492. DOI: 10.1145/2837614.2837625. URL: https://doi.org/10.1145/2837614.2837625.

[21] J. Helary, M. Raynal, G. Melideo, and R. Baldoni. “Efficient causality-tracking timestamping.” In: *IEEE Transactions on Knowledge and Data Engineering* 15.5 (2003), pp. 1239–1250. DOI: 10.1109/TKDE.2003.1232275.

[22] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming.” In: *Commun. ACM* 12.10 (Oct. 1969), 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[23] F. Houshmand and M. Lesani. “Hamsaz: Replication Coordination Analysis and Synthesis.” In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290387. URL: https://doi.org/10.1145/3290387.

[24] A. D. Kshemkalyani, A. Khokhar, and M. Shen. “Encoded Vector Clock: Using Primes to Characterize Causality in Distributed Systems.” In: *Proceedings of the 19th International Conference on Distributed Computing and Networking*. ICDCN ’18. Varanasi, India: Association for Computing Machinery, 2018. ISBN: 9781450363723. DOI: 10.1145/3154273.3154305. URL: https://doi.org/10.1145/3154273.3154305.

[25] L. Lamport. *The TLA+ Home Page*. Accessed February 2020. 2018. URL: http://lamport.azurewebsites.net/tla/tla.html.

[26] L. Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System.” In: *Concurrency: The Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, 2019, 179–196. ISBN: 9781450372701. URL: https://doi.org/10.1145/3335772.3335934.

[27] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. "Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary." In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 265–278. ISBN: 978-1-931971-96-6. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li.

[28] Lighbend. *Actor References, Paths and Addresses*. Accessed October 2020. 2020. URL: https://doc.akka.io/docs/akka/current/general/addressing.html.

[29] Lighbend. *Akka: build concurrent, distributed, and resilient message-driven applications for Java and Scala*. Accessed October 2020. 2020. URL: https://akka.io/.

[30] Lighbend. *Akka Persistence Cassandra*. Accessed October 2020. 2020. URL: https://doc.akka.io/docs/akka-persistence-cassandra/current/index.html.

[31] Lighbend. *Classic Distributed Publish Subscribe in Cluster*. Accessed October 2020. 2020. URL: https://doc.akka.io/docs/akka/current/distributed-pub-sub.html.

[32] Lighbend. *Classic Persistence*. Accessed October 2020. 2020. URL: https://doc.akka.io/docs/akka/current/persistence.html.

[33] Lighbend. *Cluster Membership Service*. Accessed October 2020. 2020. URL: https://doc.akka.io/docs/akka/current/typed/cluster-membership.html.

[34] Lighbend. *Cluster Specification*. Accessed October 2020. 2020. URL: https://doc.akka.io/docs/akka/current/typed/cluster-concepts.html.

[35] Lighbend. *Message Delivery Reliability*. Accessed October 2020. 2020. URL: https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html.

[36] Lighbend. *Persistence Plugins*. Accessed October 2020. 2020. URL: https://doc.akka.io/docs/akka/current/persistence-plugins.html.

[37] Lighbend. *Why modern systems need a new programming model*. Accessed October 2020. 2020. URL: https://doc.akka.io/docs/akka/current/typed/guide/actors-motivation.html.

[38] J. Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. "Warranties for Faster Strong Consistency." In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 503–517. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/liu_jed.

[39] T. Magrino, J. Liu, N. Foster, J. Gehrke, and A. C. Myers. "Efficient, Consistent Distributed Computation with Predictive Treaties." In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303987. URL: https://doi.org/10.1145/3302424.3303987.

[40] F. Mattern. "Virtual Time and Global States of Distributed Systems." In: *Parallel and Distributed Algorithms*. North-Holland, 1988, pp. 215–226.

[41] B. Medjahed, M. Ouzzani, and A. Elmagarmid. *Generalization of ACID Properties*. Tech. rep. Purdue University, 2009. URL: https://docs.lib.purdue.edu/ccpubs/97/.

[42] F. Meirim, M. Pereira, and C. Ferreira. "CISE3: Verificação de aplicações com consistência fraca em Why3." In: *INForum 2019 - Actas do 11º Simpósio de Informática* (2019). Universidade de Coimbra.

[43] Microsoft. *Dafny: A Language and Program Verifier for Functional Correctness*. Accessed January 2020. 2008. URL: https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/.

[44] V. Mihalcea. *A beginner's guide to ACID and database transactions*. Accessed January 2020. 2019. URL: https://vladmihalcea.com/a-beginners-guide-to-acid-and-database-transactions/.

[45] S. S. Nair, G. Petri, and M. Shapiro. "Proving the safety of highly-available distributed objects." In: *ESOP 2020 - 29th European Symposium on Programming*. Dublin, Ireland, Apr. 2020. URL: https://hal.archives-ouvertes.fr/hal-02424317.

[46] M. Najafzadeh, A. Gotsman, H. Yang, C. Ferreira, and M. Shapiro. "The CISE Tool: Proving Weakly-Consistent Applications Correct." In: *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*. PaPoC '16. London, United Kingdom: Association for Computing Machinery, 2016. ISBN: 9781450342964. DOI: 10.1145/2911151.2911160. URL: https://doi.org/10.1145/2911151.2911160.

[47] T. Pozzetti and A. D. Kshemkalyani. "Resettable Encoded Vector Clock for Causality Analysis With an Application to Dynamic Race Detection." In: *IEEE Transactions on Parallel and Distributed Systems* 32.4 (2021), pp. 772–785. DOI: 10.1109/TPDS.2020.3032293.

[48] M. Raynal and M. Singhal. "Logical time: capturing causality in distributed systems." In: *Computer* 29.2 (1996), pp. 49–56. DOI: 10.1109/2.485846.

[49] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. "The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, 1311–1326. ISBN: 9781450327589. DOI: 10.1145/2723372.2723720. URL: https://doi.org/10.1145/2723372.2723720.

[50] F. Ruget. "Cheaper matrix clocks." In: *Distributed Algorithms*. Ed. by G. Tel and P. Vitányi. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 355–369. ISBN: 978-3-540-48799-9.

[51] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: https://hal.inria.fr/inria-00555588.

[52] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-Free Replicated Data Types." In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, 386–400. ISBN: 9783642245497.

[53] F. J. Torres-Rojas and M. Ahamad. "Plausible clocks: Constant size logical clocks for Distributed Systems." In: *Distributed Algorithms*. Ed. by Ö. Babaoğlu and K. Marzullo. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 71–88. ISBN: 978-3-540-70679-3.

[54] X. Wang, J. Mayo, W. Gao, and J. Slusser. "An Efficient Implementation of Vector Clocks in Dynamic Systems." In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications, PDPTA 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 2*. Ed. by H. R. Arabnia. CSREA Press, 2006, pp. 593–599.

[55] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. "Understanding replication in databases and distributed systems." In: *Proceedings 20th IEEE International Conference on Distributed Computing Systems*. 2000, pp. 464–474. DOI: 10.1109/ICDCS.2000.840959.

[56] M. Wiesmann and A. Schiper. "Comparison of database replication techniques based on total order broadcast." In: *IEEE Transactions on Knowledge and Data Engineering* 17.4 (2005), pp. 551–566. ISSN: 2326-3865. DOI: 10.1109/TKDE.2005.54.