



Degree Project in Computer Science and Engineering

Second cycle, 30 credits

Handling of mobile applications state using Conflict-Free Replicated Data Types

ANNA TRANQUILLINI

Handling of mobile applications state using Conflict-Free Replicated Data Types

ANNA TRANQUILLINI

Degree Programme in Computer Science and Engineering

Date: August 29, 2022

Supervisors: Roberto Guanciale, Giovanni Malnati

Examiner: Stefano Markidis

School of Electrical Engineering and Computer Science

Host company: Bending Spoons

Swedish title: Hantering av mobilapplikationer med hjälp av Conflict-Free
Replicated Data Types

Abstract

Mobile applications often must synchronize their local state with a backend to maintain an up-to-date view of the application state. Nevertheless, in some cases, the application's ability to work offline or with poor network connectivity may be more significant than guaranteeing strong consistency. We present a method to structure the application state in a portable way using the Redux pattern and the properties of strongly typed languages. This method allows employing Conflict-free Replicated Data Types to create a custom converging state: this way, each replica can edit its local state autonomously and merge conflicts with other replicas when possible. Furthermore, we propose to keep a server as the communication channel and analyze how this architecture impacts design choices and optimizations related to CRDTs. Finally, we evaluate our method on a note-taking application using a few well-known CRDT designs and quantitatively justify our design choices.

Keywords

Conflict-Free Replicated Data Types, local-first, mobile

Sammanfattning

Mobilapplikationer måste ofta synkronisera lokala tillstånd med backend för att upprätthålla en uppdaterad vy av applikationstillstånd. I vissa fall kan dock applikationens förmåga att arbeta offline eller med dålig nätverksanslutning vara viktigare än att garantera strong consistency. Vi presenterar en metod för att strukturera applikationstillståndet på ett portabelt sätt med hjälp av Redux-mönstret och egenskaperna hos starkt typade språk. Den här metoden gör det möjligt att använda Conflict-free Replicated Data Types för att skapa ett anpassat konvergerande tillstånd: på så sätt kan varje replik redigera sin lokala status självständigt och slå samman konflikter med andra repliker när det är möjligt. Dessutom föreslår vi att behålla en server som kommunikationskanal och analysera hur denna arkitektur påverkar designval och optimeringar relaterade till CRDT. Slutligen utvärderar vi vår metod på en anteckningsapplikation med några välkända CRDT-designer och motiverar kvantitativt våra designval.

Nyckelord

Conflict-Free Replicated Data Types, local-first, mobile

Sommaro

Le applicazioni mobile spesso devono sincronizzare il loro stato locale con il back-end per mantenere una visione aggiornata dello stato dell'applicazione. Tuttavia, in alcuni casi, la capacità dell'applicazione di funzionare offline o con una scarsa connettività può essere più importante del garantire la “strong consistency”. In questa tesi presentiamo un metodo per strutturare lo stato di un'applicazione in modo portabile utilizzando il pattern Redux e le proprietà dei linguaggi fortemente tipizzati. Questo metodo consente di utilizzare i Conflict-free Replicated Data Types per creare uno stato convergente ad hoc: in questo modo, ogni replica può modificare il proprio stato locale in modo autonomo e risolvere i conflitti con le altre repliche quando possibile. Inoltre, proponiamo di mantenere un server come canale di comunicazione e di analizzare come questa architettura influisca sulle scelte progettuali e sulle ottimizzazioni relative ai CRDT. Infine, valutiamo il nostro metodo su un'applicazione per prendere appunti utilizzando alcuni CRDT noti e giustifichiamo quantitativamente le nostre scelte di progettazione.

Parole chiave

Conflict-Free Replicated Data Types, local-first, mobile

Acknowledgments

First of all, I want to thank my family for supporting me during my studies and pushing me beyond my limits way before university, allowing me to become the person I am today. Thanks for believing in me and supporting me during any experience I decided to partake in, even when it scared you.

Secondly, I want to thank all the friends that have been by my side during this path. I want to thank Giulia, Marcella, and Eleonora for going through the first year of university with me and making me feel at home from day one. I want to thank all the friends from the Politong project for one of the most amazing years and some of the craziest adventures. I want to thank all the people from the Madhouse for being my family for almost two years. In particular, I am grateful to Franz for being one of the best pals one could ask for and constantly pushing me to be a bit more reckless. I want to thank Nato for being the sweetest and kindest person I know, showing friendship with facts rather than words. I also want to thank Dario for being my best friend and emotional support in the most challenging moments. Furthermore, I want to thank Frank, Matte, Benna, and Mario for almost being my university classmates in Sweden and taking care of me. I want to thank the beautiful people from the Språkcafé for making me laugh and for letting me enjoy Stockholm with them. I am also grateful to all the people from Bending Spoons, especially the Mobile Technology team, who have supported me through the thesis and dedicated a lot of their time to helping me grow. A special thank goes to Lorenzo, who has guided me through this thesis and taught me important lessons regarding the field while being funny, friendly, and wise.

Finally, I want to thank Roberto, my KTH supervisor, for the help in structuring and correcting the thesis.

Stockholm, August 2022

Anna Tranquillini

Contents

1	Introduction	1
1.1	Background	1
1.2	Problem	2
1.2.1	Research questions	2
1.2.2	Original problem and definition	2
1.2.3	A note-taking application	3
1.3	Main contributions	3
1.4	Purpose	4
1.5	Outline	4
2	Background and State of the art	7
2.1	Mobile paradigms	7
2.1.1	The Context of Mobile Usage	7
2.1.2	Mobile as a distributed system	8
2.1.3	Offline-First paradigm	9
2.1.4	Local-First paradigm	9
2.2	Consistency models	11
2.2.1	CAP Theorem	11
2.2.2	Strong Consistency	11
2.2.3	Strong Eventual Consistency	12
2.3	Concurrency semantics	13
2.3.1	Happens before relationship	13
2.3.2	Total ordering of events	13
2.3.3	Logical clocks	14
2.4	Conflict-free Replicated Data Types	16
2.4.1	The interface	16
2.4.2	CRDTs types	17
2.4.3	Optimizations	18
2.5	CRDTs Portfolio	20

2.5.1	Flags	21
2.5.2	Counters	21
2.5.3	Registers	22
2.5.4	Sets	22
2.5.5	List / Sequence	24
2.5.6	Maps	24
2.5.7	Other CRDTs	25
2.6	Technical background	25
2.6.1	Redux	25
2.6.2	Type system	27
2.7	Implementations	28
3	Method	31
3.1	Specification and Technical details	31
3.2	Client-server architecture	32
3.2.1	State based vs. Operation based	32
3.2.2	Timestamps and concurrency semantics	33
3.2.3	Synchronization strategy	34
3.2.4	Garbage collection	35
3.2.5	Handling of dynamic nodes	35
3.3	Integration into codebase	36
3.3.1	Hierarchy of CRDTs	36
3.3.2	CRDT Interface	40
3.3.3	A complete flow	45
3.3.4	Composition of maps CRDTs	47
4	Results and Analysis	51
4.1	Client-Server architecture	51
4.1.1	Evaluation strategy	51
4.1.2	Benchmark on CRDT types	51
4.1.3	Benchmark on garbage collection	52
4.1.4	Number of merges	54
4.2	Integration into code base	54
4.2.1	Evaluation strategy	54
4.2.2	Benchmarks on CRDT designs	55
4.2.3	Benchmarks on note-taking application	61
4.2.4	Lines of code comparison	62
4.2.5	Discussion	64

5	Conclusions and Future work	65
5.1	Conclusions	65
5.2	Limitations	66
5.3	Future Work	66
5.4	Reflections	66
	References	69

List of Figures

2.1	CRDT boundary with Redux pattern	26
3.1	AppState hierarchy	41
3.2	Counter CRDT from literature	41
3.3	Revisited Counter CRDT	42
3.4	Reduced CRDT boundary with Redux pattern	45
4.1	Memory occupation comparison between state and operation based LWVSet	53
4.2	Comparison of AppState size with and without garbage collection	54
4.3	Register's benchmarks	57
4.4	Counters' benchmarks	58
4.5	Sets' benchmarks	59
4.6	Maps' benchmarks	60
4.7	Array's benchmark	61
4.8	Note-taking application mock execution benchmark	62
4.9	Note-taking application line of codes distribution (whole application)	63
4.10	Note-taking application lines of code distribution (AppState and Reducers)	64

List of Tables

4.1	Register's operations percent variation from replicable to wrapped scenario	57
4.2	Counters's operations percent variation from replicable to wrapped scenario.	58
4.3	Sets's operations percent variation from replicable to wrapped scenario.	59
4.4	Maps's operations percent variation from replicable to wrapped scenario.	60
4.5	Array's operations percent variation from replicable to wrapped scenario.	61

Listings

3.1	AppState	37
3.2	AppState with CRDTs	40
3.3	Literature Counter usage	41
3.4	Revisited counter usage	42
3.5	Wrapped Counter usage	44
3.6	AppState with property wrappers	45
3.7	Composed dictionary behaviour	48
3.8	Composed dictionary merge	49

List of acronyms and abbreviations

CAP	Consistency Availability Partition tolerance
CRDT	Conflict-free Replicated Data Type
EC	Eventual Consistency
HLC	Hybrid Logical Clock
HVC	Hybrid Vector Clock
LC	Logical Clock
LOC	Lines Of Code
LWW	Last-Write-Wins
P2P	Peer To Peer
PN	Positive-Negative
SEC	Strong Eventual Consistency
VC	Vector Clock

Chapter 1

Introduction

This chapter describes the specific problem that this thesis addresses, the context of the problem, the goals of this project, its main contributions and outlines the structure of the thesis.

1.1 Background

Mobile clients usually synchronize local states with a central back-end to maintain an updated view of the application state. When users interact with the same application on different platforms (i.e., multiple smartphones or tablets), they expect a consistent experience between devices. Nevertheless, a consistency issue occurs if the user can use different platforms simultaneously or offline.

Conflict-free Replicated Data Types (CRDTs) are abstract data types that can solve this consistency issue. In fact, they allow for optimistic updates and merge without any conflict. To allow for this behavior, they loosen their consistency semantics requirements and adhere to a set of mathematical properties such as commutativity, associativity, and idempotence [1].

Nowadays, most industrial CRDT use has been in distributed systems, but this technology can also be used in the context of local-first applications. While many libraries exist to support web applications in adopting CRDTs [2, 3], the same can not be said for mobile applications. In fact, the two fields adopt a different technology stack: the first is primarily based on weakly typed languages (i.e., Javascript), while the latter is based on strongly typed ones (i.e., Swift, Java, Kotlin).

1.2 Problem

This thesis investigates a method to structure ad hoc convergent mobile application states with CRDTs. Furthermore, it studies adapting and optimizing CRDTs to a client-server architecture.

1.2.1 Research questions

- **Client-server architecture:** How does a client-server architecture influence the CRDT technology?
- **Integration into codebase:** How can CRDTs be seamlessly integrated into a native mobile application? What are the benefits and drawbacks of this method?

1.2.2 Original problem and definition

This thesis project has been carried out with Bending Spoons [4]: an Italian company that develops mobile applications, founded in 2013 and based in Milan. The research questions of this thesis derive from a problem presented by Bending Spoons.

In its first years of work, the company followed a commercial strategy that led to creating and publishing a significant number of applications on the AppStore. Given the high number of products, the company used to have a single server supporting all backed operations (i.e., authentication, storage). Since the server had to support many different applications, the applications' architecture was designed to have a thick front end: most of the application logic was handled on the client. Over the years, the company's strategy has significantly shifted, leading to fewer and more dedicated products; nevertheless, the architecture of the applications has remained the same.

A common characteristic of Bending Spoons' mobile applications is that they are not content-based but provide the user with a service (i.e., training, diet coach, video editing). Most of these services are also made available offline. Consequently, different devices connected to the same profile can encounter inconsistencies between their local state and the backed-up one.

Therefore, Bending Spoons' proposal to study how to implement CRDTs into their applications to solve the consistency issues has led to the creation of this thesis.

1.2.3 A note-taking application

In order to propose clear and valuable practical examples, both when talking about functionalities and code snippets, this thesis will primarily use a single illustration. The illustration in question is a note-taking application. This subject was chosen because it is suitable for offline use and collaboration, but it is also simple and familiar enough to understand its functionalities clearly.

The imaginary note-taking application offers functionalities such as: adding a new note, removing notes, and editing them. A note contains a title, a description, a set of categories it refers to, and whether it is pinned. The application also allows a user to log in and keeps track of its name and age. Finally, the application lets the user filter notes according to categories. When trying to picture the example, the reader can imagine a product similar to *'Apple Notes'* or *'Google Keep'*. The main scenario in which we expect our note-taking application to function is where multiple devices are connected to the same user account. Devices may be simultaneously online or intermittently offline. We expect the user to modify the application's state (i.e., add, edit and remove notes) on all devices at different moments or concurrently and still end up in a consistent state once all devices have been synchronized with the server.

1.3 Main contributions

The first contribution of this thesis is the study of how client-server architecture influences CRDT technology. This analysis is made of different factors. We discuss why state-based CRDTs are the optimal choice in this context, and we use a benchmark to support the thesis that state-based CRDTs require less memory than their operation-based counterpart. We also evaluate the different types of logical clocks and explain why Hybrid Logical clocks are the best option to support both causality and relation with physical time. Furthermore, we discuss which synchronization strategies are possible and why the client-server architecture generally decreases the number of merge operations performed by the system. We also examine the advantages regarding distributed garbage collection and the handling of dynamic nodes. In fact, in both cases, a server allows to aggregate information in a central place which is beneficial both to know which replicas are part of the system and whether they are up to date.

The second contribution of this thesis is the definition of a method to build convergent mobile application states. We discuss the elements that

compose the application state and how they relate to one another. Then we propose a way to remodel the CRDT interface and suggest an optimal way to use the remodeled CRDTs in the most widespread languages for native app development. The remodeled CRDTs are also used to formalize a way to implement a map CRDT by specifying which set and CRDT behavior to adopt. Furthermore, we use a mock note-taking application to perform benchmarks and compare three scenarios: one without CRDTs, one with standard ones, and one with remodeled ones. The benchmarks evaluate the single CRDT performances, the performance of the mock application's execution, and the number of lines of code in each scenario. The results are used to conclude that while the remodeled CRDTs do lead to an increase in execution time, they also allow for increased code portability and modularity.

1.4 Purpose

This thesis project aims to research the best way to apply the CRDT technology to the problem proposed by Bending Spoons. Nevertheless, the project looks at the problem from a more general point of view and tries to provide discussion and solutions that can be useful outside of the company.

This thesis can be a valuable read for anyone who is trying to implement a solution that adopts CRDTs within a native mobile framework. Furthermore, we believe that some concepts described in the thesis, could be generally relevant to anyone that is trying to gain a knowledge and innovate in this research field.

1.5 Outline

Chapter 2 introduces the required technical background needed for this thesis. In this chapter we give an overview of relevant mobile paradigms and explain the concept of consistency model and concurrency semantics necessary to understand CRDTs characteristics. Furthermore, we describe the optimizations that have been studied in this field and offer a portfolio of CRDTs.

Chapter 3 dives into the two objectives of the thesis. Firstly, we discuss the consequence that opting for a client-server architecture has on different elements such as the choice of the CRDT type, the timestamp in use, the synchronization strategy, and the handling of garbage collection and dynamic nodes. Secondly, we propose a high-level overview of the integration into

the codebase objective. This solution combines a remodeling of the CRDT interface with the definition of the hierarchical structure of the application state.

Chapter 4 aims to present some benchmarks that can support the design choices and evaluate the performance of the solution proposed in the Methodology section. Furthermore, in this chapter we conduct an analysis and discussion of the results of the evaluation.

Chapter 5 summarizes the results and analysis as a conclusion and introduces future work to further evaluate and utilize the findings of this thesis.

Chapter 2

Background and State of the art

In this chapter, we aim to give a high-level overview of the topics needed to understand the context of the thesis. For this reason, we start by clarifying the relationship between mobile computing and distributed systems, and we describe some mobile paradigms that have been formalized over the years. We continue by introducing the concept of a consistency model and characterizing strong eventual consistency. After that, we dive deep into concurrency semantics to explain which techniques can be adopted to order events (partially or totally) in a distributed system. We continue by introducing CRDTs formally and explaining the difference between state-based and operation-based CRDTs and the optimizations that have been proposed in the research. A portfolio of notable CRDTs is also presented. Finally, we offer some background knowledge on the Redux paradigm and type systems, and we propose a few relevant implementations adopting CRDTs.

2.1 Mobile paradigms

2.1.1 The Context of Mobile Usage

Mobile devices have become a fixture of everyday life for millions of people. Across the globe, devices such as smartphones and tablets have evolved into essential tools for communication, information, and entertainment alike. The mobile market share not only has already surpassed the desktop one in 2017 but has also been steadily growing since then. In fact, in 2021, mobile devices accounted for up to 55% of the global market share.[5]

Furthermore, research shows that 88% of the time spent using mobile internet in the US is within apps, while only the remaining 12% is spent using

a mobile browser.[6]

This switch from desktop-based to mobile-based technologies has also led to a change in product requirements. For instance, in 2010, during the Mobile World Congress, the CEO of Google introduced the “mobile-first” rule and suggested designers follow this approach in product design. “Mobile-first” means starting the design and optimization process with mobile devices in mind.[7]

Nevertheless, differences between mobile and desktop do not only boil down to design patterns. Users have higher expectations for mobile apps in comparison to their web counterparts: faster speed, better responsiveness, and increased usability are expected.

Furthermore, users hope to use mobile applications even when offline or while experiencing poor network connectivity. This customer desire has sparked the need to find a better solution to access applications and web services from mobile devices, even when offline.

2.1.2 Mobile as a distributed system

Typically, when working on new technologies in a mobile scenario, we should consider the main challenges of the mobile computing context, such as power consumption, security/privacy, and connectivity/reliability. Nevertheless, this thesis looks at the problem by a higher level of abstraction and is primarily interested in the mobile user experience challenges.

Mobile devices can be seen from the point of view of distributed systems: systems with multiple components located on different machines that communicate and coordinate actions in order to appear as a single coherent system to the end-user. Nevertheless, mobile distributed systems add the complexity of being composed of dynamic nodes that can connect and disconnect freely.

In 1997 Baquero and Moura pointed out that by resorting to the traditional client-server architecture, we are not exploiting the full potential of distributed systems and mobile computing and that it is worth sacrificing consistency for allowing availability. In fact, both of these fields go against the traditional expectation of steady and continuous communication among machines. Therefore, network partitions should not be regarded as occasional faults in these scenarios but should be considered early in the design phases as they can occur frequently or during long periods.[8]

2.1.3 Offline-First paradigm

Offline-First is a software paradigm that requires software to work offline as well as it does online. As a baseline, the app is assumed not to have a working network connection with Offline-First. Then, the app can be progressively enhanced to take advantage of network connectivity when available. The paradigm leads to a shift in mindset by not associating lack of connectivity to an error condition.

Implementing a solution that follows the Offline-First paradigm requires storing data at the client-side so that the application can still access it when the internet goes away.

One possible solution is to “cache data”: data previously pulled down from remote is re-used later on. As the user encounters unstable networking situations with intermittent or no connectivity, the application will look to the local data cache to perform functions and render displays. However, if the user wanders into a part of the application they have not visited recently, the user experience is subject to the same problems as a live-data-only design. To overcome this problem, it is possible to store an entire replica of the online version; nevertheless, this solution has the drawback of requiring a much higher local memory space. If we go back to the note-taking application example, caching data would imply downloading part of the state from the server so that a user can freely read and filter all the cached notes while offline.

2.1.4 Local-First paradigm

Storing data at the client-side is fundamental to allow for offline behavior (see Section 2.1.3). However, it comes with some additional challenges: what happens if the user tries to modify the local data? In this context, the Local-First paradigm comes into play. Local-First software proposes principles that enable both collaboration and ownership for users.

Typically, the server’s data is considered the primary source of truth, while the copy on the client is seen as a cache subordinate to the server. If a modification of the local data is not notified to the server, it is not taken into consideration. In Local-First applications, these roles are swapped: the copy of the data on the local device is considered the primary copy. Servers still exist, but they hold secondary copies of the data to assist with access from multiple devices.[9]

The principles of Local-First software are:

1. **Fast:** Because the primary copy of the data is retained on the local

device, the user never needs to wait for a server request to finish. The system can handle all operations by reading and writing files on the local memory, and data synchronization with other devices happens quietly in the background.

2. **Multi-device:** Since data is kept in the device's local storage, that data must be synchronized across all devices on which a user does their work.
3. **Offline:** Local-First applications store the primary copy of their data locally, the user can read and write this data anytime, even while offline. Therefore, Local-First apps are Offline-First by design.
4. **Collaboration:** Local-First app's ideal is to support real-time collaboration that is on par with the best cloud apps.
5. **Longevity:** Local-First software enables greater longevity because the data, and the software that is needed to read and modify your data, are all stored locally.
6. **Privacy:** Local-First apps have privacy and security built in at the core: local devices store only personal data, avoiding the centralized cloud database holding everybody's data. Local-First apps can use end-to-end encryption so that any servers that store a copy of personal files hold only encrypted data.
7. **User control** - with Local-First software, personal data is stored on the device, so the user has the freedom to process his data in arbitrary ways.

In the paper "*Local-First Software: You Own Your Data, in spite of the Cloud*", Martin Kleppman et al. not only define the Local-First paradigm but also examine a wide range of existing technologies that can satisfy the Local-First ideals. After concluding that none of the examined technologies fully satisfy the Local-First ideals, they identify Conflict-free Replicated Data Types (CRDTs) as a family of distributed systems algorithms able to support the proposed paradigm.

A small dive into some background topics is required to explain better what CRDTs are and how they can be used to support the Local-First ideals. For this reason, section 2.2 introduces consistency models and the concept of eventual consistency, while section 2.3 introduces the concept of concurrency semantics.

2.2 Consistency models

A consistency model can be seen as a contract between the system and the developer who uses it. A system is said to support a particular consistency model if operations on memory respect the rules defined by the model.

This section aims to give a brief overview of consistency models by firstly introducing the CAP theorem and then describing different models relevant to the scope of the thesis.

2.2.1 CAP Theorem

In 2000, Dr. Eric Brewer gave a keynote at the Proceedings of the Annual ACM Symposium on Principles of Distributed Computing in which he laid out his famous CAP Theorem: a shared-data system can have at most two of the three following properties: Consistency, Availability, and tolerance to network Partitions.[10] In 2002, Gilbert and Lynch converted “Brewer’s conjecture” into a more formal definition with an informal proof.[11]

To better understand the CAP theorem, it is important to specify what the three properties encompass:

- **Consistency** guarantees that every node in a distributed cluster returns the same, most recent, successful write.
- **Availability** guarantees that each read or write request for a data item will receive a response.
- **Partition Tolerance** means that the system can continue operating if the network connecting the nodes has a fault that results in partitions, where the nodes in each partition can only communicate among each other.

Selecting a consistency model usually implies picking between correctness and availability. As the CAP theorem says, more synchronization will be needed if we require a higher level of consistency with our underlying data, hence decreasing the availability.

2.2.2 Strong Consistency

Traditional replication techniques strive for single-copy consistency because they want to give users the impression that they have a single, highly accessible copy of data [12].

A standard consistency model used to handle concurrent operations is called strong consistency. An informal description of the consistency model is that it always follows a sequential execution. In a distributed environment, it assigns a node as a leader, which dictates the order of operations, preventing conflicts from occurring.

However, strong consistency requires clients to constantly communicate with the leader in order to perform operations; if a node cannot reach the leader due to a network fault, its execution is stalled [13]. This imposes unacceptable constraints on mobile devices systems that, by nature, have intermittent network connectivity and must work offline.

2.2.3 Strong Eventual Consistency

If we require that applications work regardless of network availability, we must assume that users can make arbitrary modifications concurrently on different devices.[3] This means that data on different devices may diverge for a certain amount of time and that eventually, any resulting conflicts must be resolved. To address this issue, one widely implemented model is Strong Eventual Consistency (SEC).

Eventual Consistency To define SEC, simple Eventual Consistency (EC) must be first introduced. This consistency model guarantees that “*if no new updates are made to the shared state, all nodes will eventually have the same data*” [14]. Eventual consistency is especially suited in contexts where coordination is not practical or too expensive (i.e. in mobile settings) [12].

Strong Eventual Consistency Strong eventual consistency (SEC) is a model that strikes a compromise between strong and eventual consistency. Informally, it guarantees that whenever two nodes have received the same set of updates, possibly in a different order, their view of the shared state is identical, and any conflicting updates are merged automatically.[13]

By implementing Eventual Consistency, we are choosing to forfeit the strong consistency property of the CAP Theorem. In other words, we are creating an AP system: a system that selects availability and partition tolerance over consistency. The system will respond to all requests, potentially returning stale reads and accepting conflicting writes.[12] In the scope of this thesis,

the resulting inconsistencies will be handled via the use of Conflict-free Replicated Data Types.

2.3 Concurrency semantics

The updates defined in a data type may intrinsically commute or not. As we will see in Section 2.4, some CRDTs naturally converge towards the expected result independently of the order of updates. Nevertheless, for most data types, this is not the case and many concurrency semantics are possible. Therefore, we should select the most suitable one depending on the application. [15]

2.3.1 Happens before relationship

To discuss concurrency semantics, it is essential to first describe the concept of the *happens-before* relationship [16]. In this thesis we refer to this relationship with the symbol \prec .

In a distributed system $e_1 \prec e_2$, iff:

1. e_1 occurred before e_2 in the same replica
2. e_1 is the event of sending message m and e_2 is the event of receiving that message
3. there exists an event e such that $e_1 \prec e$ and $e \prec e_2$

2.3.2 Total ordering of events

The happens-before relationship can be used to partially order events. Still, sometimes we need to provide a way to deterministically define a total order among updates.

Total ordering can be obtained through the combination of clocks with a unique node identifier. Nevertheless, because of the clock skew among nodes, these tuples (clock, identifier) do not necessarily respect the happens-before relationship.

To overcome this issue different solutions have been proposed. In the next section three types of logical clocks are introduced.

2.3.3 Logical clocks

Lamport Clocks

Logical Clocks (LC) were proposed in 1978 by Lamport for ordering events in an asynchronous distributed system.[16]

Lamport brought up two valid points concerning distributed systems:

1. there is no need to keep non-interacting nodes synchronized since no difference would be observed
2. time is less important than agreement across components of the order of events in which things occur.

Furthermore, Lamport introduced Lamport Clocks by proposing an algorithm for partial causal ordering and a relative extension for totally ordering. The paper describes Lamport clocks as event counters that are incremented with every interaction. The causality relationship captured by these clocks, called happened-before, is defined based on the passing of information rather than the passing of time.

Lamport Clocks can be extended to provide total ordering. In this case, a tie-breaker value such as a derived process identifier is needed. Therefore, nodes should hold both the Lamport timestamp value and a unique process identifier. With this extension, Lamport's logical clocks prescribe a total order on the events.

Therefore, given two events A and B, if A happened before B, then A's timestamp is smaller than B's timestamp, but vice versa is not necessarily true.

$$A \prec B \Rightarrow lc_A < lc_B$$

Lamport clocks have several drawbacks: they cannot tell us if a message was concurrent and cannot be used to infer causality between events. Furthermore, they are divorced from physical time, which signifies that events can not be associated with real-time.

Vector Clocks

In 1988, Vector Clocks (VC) were proposed as an extension to Lamport clocks. These clocks maintain a vector at each node which tracks the knowledge this

node has about the logical clocks of other nodes. [17, 18]

Vector Clocks allow us to understand the ordering of events across multiple nodes; therefore, they are valuable in understanding the flow of messages in a distributed system.

Vector clocks prescribe a partial order on the events. Therefore, given two events A and B, if A happened before B, then A's timestamp is smaller than B's and if A's timestamp is smaller than B's, then A happened before B. Furthermore, two events are considered concurrent if we can neither say that A happened before B nor that B happened before A.

We can again express these properties as follows:

$$A \prec B \Leftrightarrow vc_A < vc_B$$

$$A \equiv B \Leftrightarrow (\neg(vc_A < vc_B) \wedge \neg(vc_B < vc_A)).$$

Unfortunately, the space requirement of Vector Clocks is on the order of nodes in the system, which is considered prohibitive in the context of distributed systems.

Hybrid logical clocks

As explained in the previous sections, both Lamport Clocks and Vector Clocks are disconnected from the concept of physical time. This deficiency is a significant drawback in distributed systems. However, it is an even bigger issue in mobile computing where updates can be sporadic or even wholly missing for an exceptionally long time. Therefore a user making offline updates on multiple devices in our note-taking application both in the morning and in the afternoon could find himself with the morning events overwriting the afternoon ones. In fact, according to the clocks' perspectives, the events from the devices are unrelated.

Hybrid Logical Clocks (HLC) were first introduced by Kulkarni et al. in 2014 [19] and put together the strengths of both logical clocks and physical clocks. They capture the causality relationship like Logical Clocks and enable easy identification of consistent snapshots in distributed systems, but, most importantly, they build on top of the physical clock of the system's nodes and try to tie themselves closely with physical time.[20]

Hybrid Logical Clocks are made up of two components:

- a physical component: keeps track of physical time

- a logical component: keeps track of the causality of events happening within the same physical time

Experiments show that such algorithm and clock representation is pretty resilient to both stragglers (nodes with their physical time in the past) and rushers (in the future).[19]

2.4 Conflict-free Replicated Data Types

Adopting a weak consistency model allows replicas to diverge by design. Therefore, we require a mechanism for merging concurrent updates into a common state (see Section 2.2). Conflict-free Replicated Data Types (CRDT) have been first introduced in 2011 by Shapiro et al. [1] and provide a principled approach to address this problem [15].

A CRDT is an abstract data type characterized by a well-defined interface, designed to be replicated at multiple nodes and exhibiting the following properties:

- any replica can be modified without coordinating with other replicas.
- when two replicas receive an identical set of updates, they deterministically reach the same state. This is done by using mathematically correct rules that ensure state convergence.

2.4.1 The interface

Traditionally, CRDT types can be considered to have two values: their *real value* (containing all the metadata and structures needed to implement the logic) and their *wrapped value* (the symbolic value of the abstract data type) calculated starting from the real value. Furthermore, CRDTs offer different methods that can be divided in two groups:

- **query:** the query method returns the CRDT *wrapped value*
- **update:** update methods allow modifying the CRDT *real value*

According to different papers, these two groups can be called in different ways. This thesis uses the names query and updates, proposed in a *A comprehensive study of Convergent and Commutative Replicated Data Types* [21].

2.4.2 CRDTs types

Two types of CRDT have been formerly defined: state-based and operation-based CRDTs. In principle, op-based designs are supposed to disseminate operations, while state-based designs disseminate object states.[22] Both styles of CRDTs are guaranteed to converge towards a common, correct state without requiring any synchronization.[21]

State-based CRDT

In state-based synchronization, replicas synchronize by sending their entire local state to another replica that incorporates the received state into its own state via a merge function that, deterministically, reconciles the two states. In this way, every update eventually reaches every replica, either directly or indirectly.[1]

State-based objects require only eventual communication between pairs of replicas, and updates are idempotent. Thus, replicas receiving the same update multiple times still reach the correct final state. The only requirement to guarantee updates to reach all replicas is to have a connected synchronization graph [15, 21].

Since the requirements for the communication channel are weak, state-based CRDTs make it easier to deal with an unknown number of replicas. Furthermore, they are simpler to reason about. The main drawback of state-based CRDTs is that they can become inefficient for replicas with large states because they need to transmit their whole state around the network [21].

Operation-based CRDT

In operation-based CRDTs, the execution of an operation on the replicated data is performed in two phases:[22]

- *prepare*: this is performed only on the local replica and looks at the operation and current state to produce a message that represents the operation. The message is then sent over the network to all replicas
- *effect*: once the message from a remote replica is received, it is applied locally using *effect*.

The system must ensure that operations are not lost and that each operation is applied once. If operation-based CRDTs ensure that any two concurrent operations commute, replicas can apply those operations in either order, and the outcome is the same.

Different from state-based CRDTs, operation-based replication requires reliable broadcast communication with delivery in a well-defined delivery order.[21]

Equivalence between types

It is always possible to emulate a operation-based object using the state-based approach, and vice-versa [21].

Emulation of operation-based The emulating operation based object has a *prepare* that returns the full state, and an *effect* that performs full state-merge.[21, 22]

Emulation of state-based State-based emulation of an operation-based object essentially formalizes a reliable broadcast mechanics. Each operation-based update adds a message to a set of to-be-delivered messages while merging two replicating types takes the union of the two message sets. [21]

2.4.3 Optimizations

In the last decade, research has focused on possible optimizations. When working in this field, the primary thing to optimize is memory occupation: CRDTs are memory-expensive because of the space needed to store them and the size of the messages they have to exchange.

This section presents two of the most relevant CRDTs optimizations: delta-state CRDTs and garbage collection.

Delta-state CRDTs

As explained before, two types of CRDTs exist: state-based and operation-based. The first ensures convergence by disseminating their entire state and merging it to other replicas, while the second disseminates operations assuming an exactly-once reliable dissemination layer.

Op-based CRDTs have different advantages: they are simpler to implement, occupy less memory, and exchange smaller messages. Nevertheless, they also have limitations: they require a reliable broadcasting channel, and require operations to be executed individually on each node.

State-based CRDTs, on the other hand, do not have the same constraints. However, a fundamental disadvantage of state-based CRDTs is the high communication overhead associated with transmitting the entire state.

To solve this issue, Delta-State Conflict-Free Replicated Data Types were introduced in 2014 by Almeida et al. [23]. They are a kind of state-based CRDTs, in which delta-mutators are defined to return a delta-state: a value representing the updates induced by the mutator on the current state.

The authors also propose to coalesce local operations (deltas) together into a single state update (delta group). This way, state updates can be disseminated at a lower rate allowing for a lower communication overhead.

A trivial example of a delta-CRDT is a grow-only set, where the delta is simply a singleton set containing the element to be added.

Garbage collection

Garbage collection in CRDTs is based on the concept that the metadata accumulated in replicas during their life cycle (i.e., identifiers and tombstones) becomes useless as soon as all replicas fully converge and there are no more possible concurrent operations. Therefore, it makes sense to remove all unneeded metadata to make memory usage more efficient.

In the scope of op-based CRDTs, operations that have been observed by all replicas and thus cannot have any concurrent operations ongoing are called “causally stable”. Nevertheless, determining if an operation has reached causal stability is not trivial.

Baquero et al.[24] proposed to rely on a Reliable Causal Broadcast middleware which tracks causal information for all messages transmitted in a system. They define causal stability as follows:

“A timestamp T , and a corresponding message, is causally stable at node i when all messages subsequently delivered at i will have timestamp t greater than T .” [24]

This simple definition leads to two problems. The first issue is that a few nodes may not yet be aware of an operation’s causal stability state, whereas others are. Because causal stability is used to delete superfluous meta-data, a node unaware of an operation’s causal stability may receive an operation that is missing meta-data. Baquero et al. solve this by ensuring the RCB middleware buffers the operation until the node knows the causality state [24].

The second problem is the causal stability of an operation can only be determined if and only if every other node sends a message after they receive the operation. If one node does not issue any operations, no causal stability can be determined at any node.

Furthermore, Baquero et al. [24] also assume a fixed network where every network replica is known at setup. This is a realistic scenario in some contexts but limits the applicability of CRDTs in many modern distributed applications which are collaborative or require that the number of replicas can grow dynamically.

For these reasons, in the paper “*Memory efficient CRDTs in dynamic environments*”, Bauwens et al. [25] propose a solution that allows new nodes to join the network and construct a proper state for their local replica and perform eager garbage collection by broadcasting information regarding the casual stability of nodes.

Another possible direction to implement garbage collection is to study adding small doses of synchronization to support infrequent, non-critical client operations, such as committing a state or performing a global reset. [21] This may seem counterintuitive since it requires blocking synchronization, which is the limitation CRDTs are addressing in the first place. Nevertheless, synchronization between nodes implies cleaning all metadata accumulated until that moment. Sporadically performing strong synchronization may be a good compromise that shortly scarifies the system’s availability for more efficient memory occupation.

2.5 CRDTs Portfolio

As mentioned before, the formal definition of Conflict-free Replicating Data Types dates back to 2011[1], but since then, additional work has been done to collect and formalize different CRDT designs. In particular, this section strongly references two papers: *A comprehensive study of Convergent and Commutative Replicated Data Types* by Shapiro et al. [21] for the presentation of the different designs, and *Conflict-free Replicated Data Types: An Overview* by Prego et al. [15] for the discussion on concurrency semantics.

This section does not aim to specify the implementation details of each CRDT, but rather tries to give a general description of the CRDT in question. Suppose the reader is interested in the actual pseudo-code definition of each CRDT. In that case, we suggest reading *A comprehensive study of Convergent and Commutative Replicated Data Types*[21] where all relevant state-based and operation-based CRDTs algorithms are proposed.

The portfolio aims to show how different concurrency strategies can be chosen to allow different conflict resolution behaviors and it helps understand the challenges, possibilities, and limitations of CRDTs.

All designs that refer to the Last Writer Wins strategy are expected to define a total ordering of events according to what was specified in section 2.3.2.

2.5.1 Flags

The flag is a data structure holding a boolean value. The value represents disabled or enabled state. In the context of CRDT, the flags are not commutative since (enable, disable) and (disable, enable) doesn't mean the same.

Because of the not-commutative character we distinguish 2 types of flags:

Enable-Wins Flag

Two concurrent disable and enable always leave the flag in enable state.

Disable-Wins Flag

It's the opposite of the above, the disable flag is always left when 2 concurrent writes happen.

2.5.2 Counters

The general idea of a Counter is to keep track of a numeric value that can be either increased or decreased by one unit. Because updates are inherently commutative, a counter CRDT's natural concurrency semantics is to have a final state representing all executed updates' effects.[15]

Positive-Negative Counter

Positive-Negative counters keep track of all increments and decrements applied to them at each node. Therefore, the value of the counter can be computed by subtracting the number of decrements from the number of increments:

$$\{inc \mid inc \in O\} - \{dec \mid dec \in O\}$$

Grow-only Counter

As the name suggests, a Grow-only Counter can only be incremented. This data type can be seen as a simplification of the Positive-Negative Counter.

The counter value can be computed by counting the number of increments.

$$\{inc \mid inc \in O\}$$

Bounded Counter

A Bounded Counter is a counter where a particular constraint is specified. The constraint limits the counter from going above or below a certain threshold. The bounded counter prevents its value from violating this constraint by limiting the number of operations that can be executed in each replica before synchronizing the changes with other replicas. To do this, the bounded counter stores the number of decrement/increment operation executions available in each replica. Furthermore, replicas are allowed to transfer the availability of operations between each other.[15]

2.5.3 Registers

A register maintains an opaque value and provides a single update: $wr(value)$. Two concurrency semantics have been proposed leading to two different CRDTs: the multi-value register and the last-writer-wins register.

Multi-value Register

The Multi-value Register CRDT keeps track of all concurrently written values. Trivially, the read operation returns the set of concurrently written values.

Last-write-wins Register

In the Last-write-wins (LWW) Register CRDT, only one value is kept, if any: the last written value.

2.5.4 Sets

A set is a collection of unique elements. A set data type provides two updates:

- $add(e)$, for adding element e to the set
- $rmv(e)$, for removing element e from the set

In this case, many concurrency semantics are possible when a concurrent add and remove must be handled:

- Add-wins: the element will belong to the set.
- Remove-wins: the element will not belong to the set.
- Last-write-wins: the element will be in the set if the add is ordered after the remove in the *total order* among updates.

Grow-Only Set

The simplest solution is to avoid removals altogether. A Grow-Only Set (G-Set) supports operations add and lookup only and, by design, does not require the definition of a concurrency semantic. The G-Set is useful as a building block for more complex constructions.

Two-Phase Set

The Two-Phase Set (2P-Set) is a Set in which an element can be added and removed but never added again. The Two-Phase Set combines an adding G-Set with a removing G-Set, the latter of which is known as the tombstone set. To avoid anomalies, an element can only be removed if, for the replica, it is in the set.

The lookup operation returns elements that have been added but not been removed yet. Adding or removing the same element twice, as well as adding an element that has already been removed, has no effect.

Positive-Negative Set

The Positive Negative Set (PN-Set) associates a counter to each element, initially set to zero. Adding an element increments the associated counter, and removing an element decrements it. The element is considered in the set if its counter is strictly positive.

Observed-Remove Set

Observed Remove Set (OR-Set) uses unique tags instead of timestamps. For each element in the set, a list of add-tags and a list of remove-tags are maintained. An element is inserted into the OR-Set by having a new unique tag generated and added to the add-tag list for the element. Elements are removed from the OR-Set by having all the tags in the element's add-tag list added to the element's remove-tag (tombstone) list. To merge two OR-Sets, for each element, let its add-tag list be the union of the two add-tag lists, and likewise

for the two remove-tag lists. An element is a member of the set if and only if the add-tag list is a superset of the remove-tag list.

2.5.5 List / Sequence

A list (or sequence) data type can be used to maintain an ordered collection of elements and allows two updates:

- $ins(i, e)$ - insert element e in position i , shift element in position i , if any, and subsequent elements to the right
- $rmv(i)$ - remove element in the position i , if any, and shift subsequent elements to the left.

The design of an ordered sequence is one of the main research focuses in the field of CRDTs: this is because sequences are what hides behind collaborative text editing algorithms. Different solutions have been proposed, such as Logoot/LSeq[26], RGA[27], TreeDoc[28], etc. The main idea behind these designs is to keep a graph or tree data structure to keep track of tombstones or unique ids.

2.5.6 Maps

A map is an object that maps keys to values, the keys are unique and each key can map to at most one value. Maps export two updates:

- $put(k, o)$, associates key k with object o
- $rmv(k)$, removes the mapping for key k , if any

In this case, different concurrency semantics are possible in case of a concurrent remove and update associated with the same key.

- Remove-as-recursive-reset: a reset update is performed on object o associated with k , and recursively in all objects embedded in o . If we were to choose this semantic, we should define a reset update for each CRDT: a function that sets the object's value to a bottom value.
- Remove-wins: gives priority to removes over updates
- Update-wins: gives priority to updates over removes.

Some references [29, 30] have pointed to the fact that handling the keys of a map leads back to implementing a Set specification. In the following paragraphs, we build further on this concept.

Map of literals

Map of literals are maps that can only contain literals, namely, non-CRDT elements. In this case, we must devise the concurrency semantics to answer two questions. First, what is the value associated with a key k in the presence of two concurrent puts for k . This question can be answered by tracing it back to the semantics of registers. The second question is how we should handle a concurrent removal and put for the same key. Similarly, we can answer by associating this concurrency event to the semantic of sets. As a matter of fact, a set can be seen as a simplified map where the key is the value itself. Therefore, specifying the concurrency semantics of a map boils down to specifying the semantic of a set and register.

Map of CRDTs

Maps of CRDTs are maps whose entries are CRDT elements. In this case, the put and remove updates can still be traced back to the semantic of sets. On the other side, the value associated with a key k in the presence of two concurrent puts for k should instead rely on the semantics of the embedded CRDT. Therefore, specifying the concurrency semantics of a map boils down to specifying the semantic of a set and the embedded CRDT.

2.5.7 Other CRDTs

Several other CRDTs have been proposed in the literature, including CRDTs for elementary data structures, such as graphs, more complex structures, such as JSON documents[3], and for collaborative text editing [26].

Even if these topics are highly relevant in the research area of CRDTs, they fall out of the thesis's scope and, therefore, will be no further discussed in this work.

2.6 Technical background

This section introduces a few technical topics necessary to define the thesis's scope.

2.6.1 Redux

Redux is a pattern for managing and updating application state, using events called "actions". It serves as a centralized store for the state that needs to be

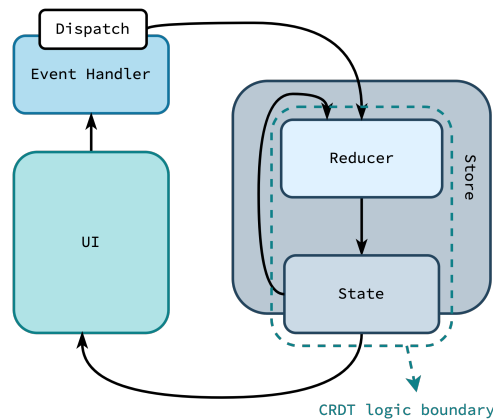


Figure 2.1: CRDT boundary with Redux pattern

used across the entire application, with rules ensuring that the state can only be updated predictably.[31]

- **Single source:** the logic of the state is not scattered around in different components, but it is kept in a single object: the AppState.
- **State is read-only:** not every function in the app can update the state; instead, a consistent mechanism is used to perform a state-update.
- **Only pure function change the state:** state changes are obtained through functions, called reducers, that take the current state and an action and return the next state.

CRDTs and Redux

In this section we examine why the Redux pattern lends itself very nicely to CRDT implementations.

In Redux, dispatched actions are applied to a state, resulting in a new state. Actions are always applied in order of their timestamp. Therefore, Redux makes it easy to define the state’s boundary and identify which operations should be considered state changes and therefore be kept in the “memory” of CRDTs.

In figure 2.1 a diagram of the Redux pattern is shown to make explicit which part of the architecture should be aware of the CRDT logic. In fact, not only should the state be able to implement the correct logic, but reducers should be aware of which operations must be performed on the type.

It is relevant to note that Bending Spoons does not precisely use Redux but develops its applications using a proprietary framework called Katana[32],

which is strongly inspired by the Redux framework. Nevertheless, in this thesis, we will always refer to the Redux framework for the sake of generalization.

2.6.2 Type system

A type system is a logical system in which a type is assigned to each language construct, such as variables, expressions, or modules. Type systems are frequently specified within programming languages and built in interpreters and compilers. The primary purpose of the type system is to define the interfaces between different parts of the codebase clearly and, therefore, reduce bugs and check whether the parts have been put together consistently. Type checking is the process of verifying and enforcing the constraints of types: this operation may occur at compile-time (a static check) or at run-time.

Static and dynamic typed language

In statically-typed languages variables' types are known already at compile-time, while in dynamically-typed languages they are known at run-time.

Strongly and weakly typed language

A strongly-typed language is one in which variables are bound to specific data types. Type errors will occur if the types in the expression do not line up as expected – regardless of when type checking occurs. Variables in weakly-typed languages, on the other hand, still have a type, but type safety constraints are less stringent than in strongly-typed languages.

Characteristics of mobile frameworks' languages

When talking about native mobile development, the most commonly used languages are:

- Swift - iOS
- Kotlin - Android (official language for Android app development)
- Java - Android (most used language for Android app development)

For the scope of this thesis, it is essential to note that each of these languages is considered both statically typed and strongly typed.

Type systems and CRDTs

When integrating Conflict-free Replicated Data Types into the code-base, a decision should be made on who should handle the replicating data logic. Two possible options are available; choosing between them mainly depends on the properties of the languages and framework in use.

1. **Weakly typed languages** - The first solution has been proposed by Martin Kleppmann et al. [3]: the idea is to arbitrarily nest lists and maps and other leaf data types, which can be modified by insertion, deletion, and assignment. In this case, each *object* takes care of handling its logic.
2. **Strongly typed languages** - The second solution, proposed by this thesis, assigns each property a specific type and semantic behavior (ex: Integer - Counter). In this case, each *type* takes care of handling the logic.

The work done in *A Conflict-Free Replicated JSON Datatype* [3] is in Javascript: in this language, maps also represent classes/objects with their properties. Furthermore, properties can be added and removed freely from an object and are not bound to a specific data type: specifying a type for each property is not possible.

On the other hand, applying the first solution to a statically and strongly typed language is impossible because the structure of the state and the type of each property should be specified from the start.

2.7 Implementations

The CRDTs concepts can be used in the scope of different applications. Some examples of relevant implementations in the field are:

- General-purpose libraries
 - Automerge[33]: a JavaScript CRDT implementation with a JSON data model.
 - Yjs[34]: a modular framework for building collaborative applications on the web.
- Distributed databases
 - The Riak database[35]: one of the first to add CRDT support in 2013 (implemented in Erlang).

- Ditto[36]: uses CRDTs for data sync between mobile devices.
- Text editors
 - crdt.el[37]: a real-time collaborative editing environment for Emacs using Conflict-free Replicated Data Types.

Nevertheless, probably the most relevant implementation for this thesis is the one proposed by Figma. Figma's technology is inspired by CRDTs, but since Figma is centralized, they have simplified their system by removing this extra overhead and benefit from a faster and leaner implementation. Figma's data structure is not a single CRDT; however, it has been inspired by multiple separate CRDTs and uses them in combination to create the final data structure that represents a Figma document[38]

These concepts are incredibly close to what this thesis aims to do. Nevertheless, Figma is a proprietary product, and even if they offered good insights on their technology in the article *How Figma's multiplayer technology works*, none of their work is open source.

Chapter 3

Method

In this chapter, we dive into the two different objectives. Firstly, we discuss the influence that opting for a client-server architecture has on different elements such as the choice of the CRDT type, the timestamp in use, the synchronization strategy, and the handling of garbage collection and dynamic nodes. Secondly, we propose a high-level overview of the integration into the codebase objective. This solution combines a remodeling of the CRDT interface with the definition of the hierarchical structure of the application state. This chapter does not contain the description of the evaluation strategy, which can be instead found in Chapter 4 together with the results.

3.1 Specification and Technical details

Programming language The work done in this thesis has been conducted by using the Swift programming language. Considerations and algorithms will be kept as general as possible, but all examples will be proposed in Swift.

Server The server used is Firebase Realtime Database: a cloud-hosted database. It only plays the role of storing the state and does not implement any additional logic. Its native API for performing transactions (atomic operations) will be used to update the remote state. Nevertheless, using this technology is not mandatory: a similar functionality can be implemented with any server by having the client send a “compare and set” operation.

Firebase Realtime Database does offer native offline support, which implements conflict resolution through a last-writer-win strategy. Since the main focus of the thesis is solving the problem in a more refined way, this functionality will not be used.

3.2 Client-server architecture

The primary context of Conflict-Free Replicated Data Types research is distributed systems. These abstract data types are suitable for use in this context because they do not require a single source of truth (i.e., a server), which is not present by design in distributed systems. On the other side, mobile devices can also be regarded as distributed system nodes because they maintain a local copy of data, which can be modified but must look consistent between devices (see Section 2.1.2). Nevertheless, differently from traditional distributed systems, using a server as a communication channel in the mobile context is a viable option. Therefore, we aim to discuss how opting for a client-server architecture influences Conflict-Free Replicated Data Types. In our opinion, the research area has neglected to investigate the advantages and optimizations that client-server architecture can bring to this technology.

3.2.1 State based vs. Operation based

This section examines how the two different CRDTs types (see section 2.4.2) can be applied to a client-server architecture and justifies why we have chosen to adopt state-based CRDTs to implement our work.

Operation based Applying operation-based CRDTs to a client-server architecture would mean following a strategy similar to the one proposed by Shapiro et al. [21]. when describing the emulation of state-based CRDTs. Each replica would send its operations to the server, which would keep a set of messages that should be forwarded to all replicas that have not yet received them.

This solution has primarily one advantage: the reduced size of messages sent across the network. The size of the remote state would be bigger than the one in the state-based solution, and the memory complexity of checking which messages have not been received by each replica could be a bottleneck.

A new device joining the network would have to download and apply all the messages to their initial state. For this reason, in a dynamic system, no message could ever be deleted (outside of garbage collection) since new nodes would need to see the system's entire history.

State based State-based CRDTs are the most obvious choice for the given context. In this case, we would keep the entire state saved on the server, and each replica would download the remote state, merge it with the local one

and re-upload the resulting state; these operations should be performed in an atomic fashion.

This choice has multiple consequences:

- The server would act as an accumulator, putting together all the states received until that moment. Therefore, the number of messages sent across the network would decrease, making the size of the messages less relevant.
- The data stored on the server would be smaller than in the operations-based solution.
- A new device connecting to the system would only need to download the remote state to be up to speed with the others.

For these reasons, we decided to base our work on state-based CRDT.

3.2.2 Timestamps and concurrency semantics

There are many strategies to order events in distributed systems (see Section 2.3). The first option is to provide a partial order of events and select a concurrency semantic to solve conflicts between concurrent updates. In this case, we will need to select a clock (i.e., the vector clock) that allows us to detect concurrent events.

The second option is to provide a total ordering of events by adding to the clock a unique identifier that allows us to solve conflicts deterministically. We can pick between any clock in this case, but the most suitable options are Lamport clocks or Hybrid logical clocks. In this case, choosing a vector clock would be overkill since we do not use the information concerning concurrency that this logical clock provides.

When making this choice, it is fundamental to consider the context: we expect nodes (i.e., devices) to be offline and edit their local copy for an extended time. This scenario would lead to a considerable number of events being considered 'concurrent'. Therefore, opting for a clock wholly detached from the concept of physical time can be dangerous. Hence, this work suggests using Hybrid Logical clocks as the timestamp provider.

Casual information is necessary for some CRDTs (i.e., Multi-value Register, Flag), which lose their meaning without it. In these cases, we may decide to combine a Hybrid Logical Clock with a Vector Clock, therefore adopting a Hybrid Vector Clock (HVC).

3.2.3 Synchronization strategy

Selecting state-based CRDTs in combination with a client-server architecture has the advantage of decreasing the number of merges performed by replicas. In fact, without a server, each replica would broadcast to other replicas a copy of its state. In the best-case scenario, namely a fully connected mesh, all replicas should merge the received state with their local one. In other topologies, replicas should instead keep disseminating the received state to ensure all nodes receive it.

By offering a client-server architecture, replicas only need to communicate with a single actor that accumulates all the information received from the other nodes. Nevertheless, it is up to us to decide when the communication between clients and servers should occur. We can select different events to trigger a client-server synchronization:

- at every local state change
- at every remote state change
- periodically (ex: every five minutes)
- forced by the user

While we usually want to keep collaborative software as strictly synchronized as possible, the same design requirement may not hold for different kinds of products. Considering our note-taking application example, we may be interested in synchronizing the behavior between devices once in a while without making the user feel like the application's state is changing under his nose. Nevertheless, updating the remote state with the local changes does not imply that we also have to apply the remote changes to the local state. Therefore, we could decide that some events may trigger a remote state update, leaving the local state unchanged. This is especially meaningful when synchronization is triggered by local changes or a periodical update.

Therefore, selecting how and when to synchronize requires considering the user's experience we want to provide. Naturally, a diverse number of merges will be performed depending on the synchronization strategy. Nevertheless, as long as we do not merge multiple times with the same remote state, which can be avoided by checking the state id, we can be sure that we will not surpass the complexity upper bound given by the serverless peer-to-peer scenario.

3.2.4 Garbage collection

Garbage collection is one of the most relevant optimizations that can be applied to a CRDT solution, because of their characteristic monotonic increase in the size (see Section 2.4.3).

Garbage collection is difficult to implement, but client-server architectures make it easier. The primary way to perform a complete garbage collection is to achieve consensus: the unanimous agreement between all replicas. By using the server as a communication channel, we know that the server will store the most up-to-date information that the replicas have shared, and we can keep track of who has seen that information.

We can store a table that monitors which replica has downloaded the most recent server state. If a replica alters the remote state, the table is refreshed. This way, once all replicas are up-to-date with the remote state, all metadata can be removed. A trivial but relevant consequence is that no metadata will be stored as long as only one replica is present in the system.

This simple solution does have significant limitations. The most evident is that all devices must download the remote state without modifying it to trigger garbage collection. Furthermore, even if the remote state has been cleaned of all metadata, we need a way to let all replicas know that they should perform garbage collection too. If we do not achieve this goal, all removed metadata will be re-introduced in the state as soon as a replica performs a merge. Because of this limitation, it may be reasonable to try forcing garbage collection when detecting a stable remote state.

3.2.5 Handling of dynamic nodes

One of the most significant issues regarding dynamic distributed systems like the one in question is the lack of knowledge regarding the elements present in the system. Generally, a gossip protocol is used to disseminate information regarding the system's state. Nevertheless, having a client-server configuration makes it much easier to navigate this issue. We can easily keep track of all nodes (i.e., devices) that have communicated with the server and remove them from this list if they notify us of their departure.

In the scope of mobile applications, shutting down a node indicates the explicit removal of the application from a device with all its local data. However, it is impossible to know when this happens and notify the server accordingly. Therefore, a criterion must be used to discern inactive nodes from nodes that have definitively left the system.

Furthermore, knowing how many and which nodes compose the system is fundamental for performing garbage collection because it relies on reaching a consensus between all nodes (see Section 2.4.3).

The simplest way to do this is to set a time threshold. The threshold could be different depending on the nature of the application. Nevertheless, we should decide how to behave when a longly-disconnected node reconnects. As a matter of fact, if we have performed garbage collection, states will possibly not be merged correctly.

Therefore, it could be relevant to determine whether we should consider outdated information. It may be meaningful to ignore metadata produced before the threshold to avoid merge errors due to garbage collection and because we may generally deem such data irrelevant.

3.3 Integration into codebase

One of the reasons that have allowed CRDTs to be adopted thoroughly in web based technology in the last years is the development of libraries such as `automerger` [33] and `Yjs` [34]. Both libraries are implemented in Javascript and offer a collection of data structures that can be used for building collaborative applications. The flexibility of languages such as Javascript allows the developer to manipulate variables without knowing their type. Nevertheless, languages typically used in native mobile development differ radically from Javascript in their characteristics. Variables are statically and strongly typed, and the state structure is fixed. Therefore, studying how to adopt CRDTs with a mobile stack is radically different from doing the same within a web development framework. Therefore, in this section, we examine how to integrate CRDTs seamlessly in a general native application.

The concepts explained in this section have been ideated and discovered while implementing a library supporting CRDTs in Swift. This library will not be made public with the thesis and is left as a property of Bending Spoons.

3.3.1 Hierarchy of CRDTs

This section describes how to compose different elements to build an ad hoc app state within the Redux pattern while adopting a strongly typed language. To achieve this goal, it first introduces the building blocks of a generic app state and then explains how to piece them together.

The building blocks

An excellent place to start is examining the state's different elements, namely basic data types (i.e., strings, integers), optional properties, collections, and nested structures.

It is essential to also keep in mind that some properties may be considered replicable, while others (that possibly only matter locally) may not.

We will consider a specific state example to examine the problem space thoroughly. The app state in listing 3.1 comes from the note-taking application example (see Section 1.2.3) and offers a complete example that comprehends the different elements. `Category` can be considered by the reader as an Enumerated Type (enum), but it is not shown in the listing because it falls outside the scope of the example.

```
struct AppState {
    var notes: [UUID: Note]
    var selectedCategories: Set<Category>
    var user: User?

    struct Note {
        var title: String
        var description: String
        var categories: Set<Category>
        var pinned: Bool
    }

    struct User {
        var name: String
        var age: Int
    }
}
```

Listing 3.1: AppState

Non-replicable Properties Non-replicable properties are properties that are not replicated across nodes. These properties usually express the execution state of the application. For instance, `selectedCategories` is a property that works as a filter: it states which categories have been chosen by the user to be displayed. It is not meaningful to replicate this concept across devices because two devices displaying different filtered notes are not experiencing a

conflict. When merging two app states, non-replicable properties should not be considered and should remain unchanged after the merge.

Replicable Properties When deciding how to handle basic replicable properties, we have to determine the CRDT associated with the semantic behavior of the property.

In the example, we can take in consideration the pinned value of each `Note`. We may pick a Register CRDT if we want its value to be the last value set by a replica. Instead, we may choose a Flag CRDT if we want the status to be pinned if at least one replica has marked it as pinned.

It is relevant to note that only a subset of CRDTs can be used to map the behavior depending on the original type of the property: trivially, a Counter can not be used to express a String property.

Collections Similar to what was done for basic data types, we associate a relevant CRDT to each collection.

For instance, `Note.categories` expresses the set of categories associated with each note. We should pick a Set CRDT (i.e., `LWWSet`, `ORSet`, `PNSet`) to handle the behavior of this property.

Nevertheless, the map CRDT can contain CRDT entries itself (see Section 2.5): in this case, map CRDTs are responsible for correctly handling and merging their entries. An example of this is the `notes` property of `AppState`. This property is a map of all notes available in the application; therefore, if a replica edits a note associated with a particular id, we want the change to be reflected in the note associated with the same id in the other replicas.

Option Types Option Types are types that represent the encapsulation of an optional value. In the library implemented by Martin Kleppmann et al. [3] objects are handled using the logic of a map and therefore have the natural ability to add and remove properties. In this thesis, given the properties of the languages under consideration, only properties marked as optional can be added (given a value) and removed (set to null).

The concept of having optional properties implies the idea of a hierarchy between replicable types. In fact, if the property has a value, its internal implementation should handle the logic. On the other side, if the property is set to null, the absence or presence of the property overpowers the internal logic.

Consequently, we should answer the questions: what happens when the property is set to null? Moreover, who should be responsible for handling this logic?

From the standpoint of this thesis, we can see being an Option Type as a CRDT behavior itself; in fact, we can use a Register CRDT to represent the optional behavior. When the value goes from none to some and vice versa, the Register will handle the logic, while when the value changes without reverting to null, the encapsulated replicable will handle it.

Structures App states are a composition of different structures, each possibly containing properties with associated replicable behavior. A structure containing at least one replicable property should be considered a replicable element. In this case, the element does not implement a specific logic but instead manages its replicable properties. For instance, merging a CRDT structure with another implies merging all the replicable properties of the two structures.

An example of this is the user property. If we consider its properties CRDTs, the user structure must contain the logic to handle it correctly. Therefore, the AppState will have to handle user not too different from what it does with other replicable properties.

Putting it all together

We can derive from the previous section that there are three types of elements that can create a hierarchical structure of replicables:

1. **Structures** - each structure instantiation handles its replicable properties
2. **Maps with replicable entries** - each map instantiation handles its replicable entries
3. **Optional properties** - each optional instantiation handles its replicable value

If we go back to the note-taking app example, we can rewrite the state by associating to each property a CRDT. In listing 3.2 we can see a potential state declaration.

```
struct AppState {
  var notes: ReplicableDictionary<UUID, Note>
  var selectedCategories: Set<Category>
  var user: OptionalReplicable<User>

  struct Note {
    var title: ReplicableRegister<String>
    var description: ReplicableRegister<String>
    var categories: ReplicableSet<Category>
    var pinned: ReplicableFlag<Bool>
  }

  struct User {
    var name: ReplicableRegister<String>
    var age: ReplicableRegister<Int>
  }
}
```

Listing 3.2: AppState with CRDTs

In this case the AppState hierarchy will work as follows:

- AppState will manage its replicable properties: notes and *optional of user*
- the *optional of user* will manage user
- user will manage its replicable properties: name and age
- notes will manage all of its *note entries*
- each *note entry* will manage its replicable properties: title, description, categories and pinned

In figure 3.1 we can see a diagram representing the above described hierarchy.

3.3.2 CRDT Interface

An interface defines the operations supported by a data structure and those operations' semantics or meaning. This section explores how CRDTs interfaces are described in the literature and how they can be modernized to make their use more flexible and clean.

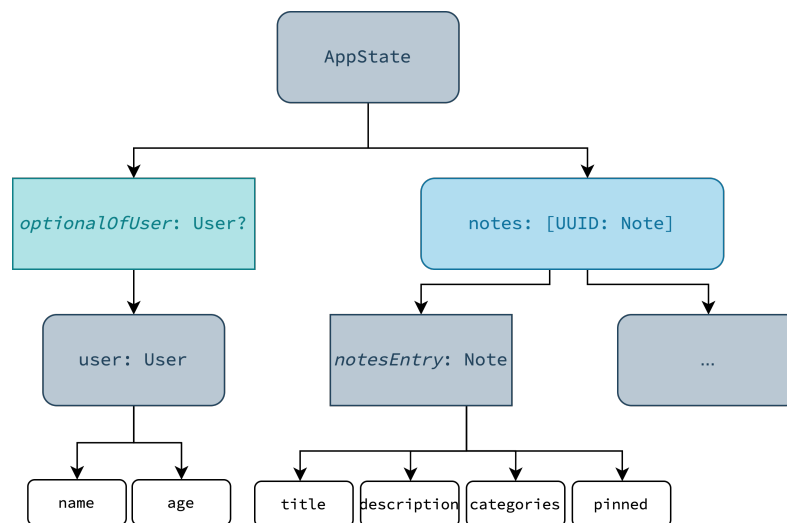


Figure 3.1: AppState hierarchy

The standard interface

Generally, a CRDT interface offers access to two values (the *real value* and the *wrapped value*) and a series of functions: a query and a few update methods (see Section 2.4.1).

In figure 3.2 and listing 3.3 we can look at the standard interface of a Counter CRDT and its usage. The counter has a real value made up of different properties such as maps and ids, while its wrapped value is a single numeric value obtained by applying the query method to the real value. Furthermore, the counter interface contains the query method itself (`getWrappedValue()`) and two update methods to increase and decrease its value.

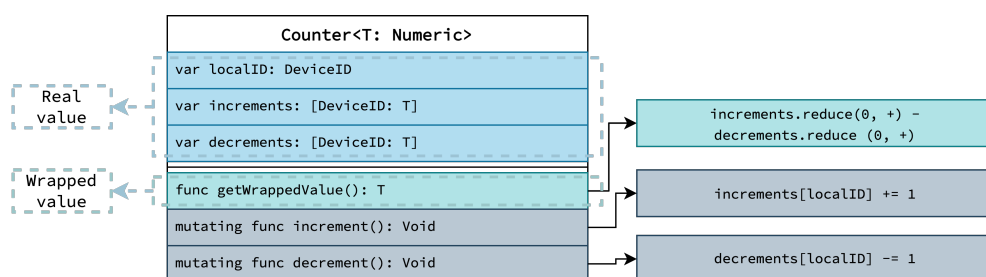


Figure 3.2: Counter CRDT from literature

```
var counter: Counter<Int>(0)
```

```
// increase
counter.increase() // wrapped value = 1
// decrease
counter.decrease() // wrapped value = 0
```

Listing 3.3: Literature Counter usage

Remodelling

CRTDs try to look and act as their wrapped value from the outside while simultaneously hiding additional data and computations. If we want to further this idea, we could combine the update methods into a single one. In fact, we can build a single update method that modifies the real value by specifying the desired wrapped value. We call this function `setWrappedValue`.

The `getWrappedValue` / `setWrappedValue` pair can be more elegantly implemented with a computed property: a property where we declare custom getters and setters. Nevertheless, for the clarity of this thesis, we will keep referencing these methods explicitly.

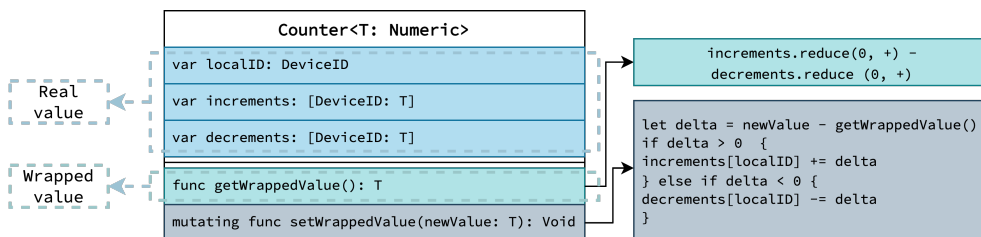


Figure 3.3: Revisited Counter CRDT

```
var counter: Counter<Int>(0)

// increase
counter.setWrappedValue(newValue: 1) // wrapped value = 1
// decrease
counter.setWrappedValue(newValue: 0) // wrapped value = 0
```

Listing 3.4: Revisited counter usage

For instance, we can transform the Counter structure from figure 3.2 to figure 3.3. In this case, if a Counter's wrapped value is equal to 2, and we set its new wrapped value to 5, this will be translated to an increase of three units.

This idea, naturally, can be applied to all CRDTs, as long as we can devise a function that can translate an initial wrapped value - final wrapped value pair into a series of proper edits. While for most CRDTs this function can be trivially written, this is not true for all. The following paragraphs discuss the challenges of adopting the remodeled interface.

Invalid operations Not all CRDTs allow all operations to be performed on them. Notably, the Grow-only Counter and Add-only Set respectively allow only to be increased or enlarged. Therefore, we should decide how to handle situations where these CRDTs are set to an invalid value. We have three main options: fail, rollback, or perform a partial operation.

We can use the example of an Add-only Set to explain the three options. We have an Add-only Set with initial value $[a, b]$ and we set its new wrapped value to $[a, c]$: this translates to removing b and adding c . If we choose to fail, we should ask the program to abort after detecting a removal. Instead, if we choose to roll back, we should return from the function without editing the real value. Finally, if we choose to perform a partial operation, we should add c to the real value before returning.

It is evident that by performing rollback or partial operations, we allow the final wrapped value to be different from the one we specified. For this reason, this work opts for the first option. Failing also allows the system to let the developer know that the wrong CRDT was associated with a specific property. This gives the developer the possibility to correct his choice. Performing rollback or partial operations, instead, could lead to unexpected bugs.

Shortest edit script in list CRDTs Implementing the `setWrappedValue` function is not always as straightforward as in the Counter example, and it can lead to additional complexity. In fact, when working with list/sequence CRDTs, we need a way to determine a sequence of edits that can lead us from the initial state to the final state.

A straightforward solution to this problem is to delete all elements of the initial state and then insert each element of the final state in the sequence. However, we would not consider this a good-quality solution since it does not let our CRDT behave as it should: a function that replaces the entire sequence is not of much use.

Thankfully, this is not a new problem in computer science. Myers Difference Algorithm (MDA) [39] is an algorithm that finds the longest common subsequence (LCS) or shortest edit scripts (SES) of two sequences. The common subsequence of two sequences is the sequence of elements that

appear in the same order in both sequences. Therefore, the Myers algorithm can be used to determine a sequence of changes that should be applied to sequence A to become sequence B.

Wrapped CRDTs

This thesis proposes one last step towards code portability, which is highly dependent on the programming language in use. In fact, this last step is based on a programming language concept called property wrappers (Swift [40]) or delegated properties (Kotlin [41]).

When dealing with properties representing some form of state, it is widespread to have some associated logic that gets triggered every time a value is modified. For example, we might validate each new value according to a set of rules, transform our assigned values somehow, or notify a set of observers whenever a value is changed. Property wrappers and delegated properties add a layer of separation between code that manages how a property is stored and the code that defines a property.

Using this concept, we can define our properties with their original type and add a wrapper/delegate to implement the replicable logic. The wrapper/delegate will overwrite the getter and setter of the property with the `getWrappedValue` and `setWrappedValue` functions. This way, the property will be used as a simple property while simultaneously encapsulating the CRDT logic.

In listing 3.5, we can see how these types of properties can be used.

```
@Counter var counter: Int = 0

// increase
counter = 1 // wrapped value = 1
// decrease
counter = 0 // wrapped value = 0
```

Listing 3.5: Wrapped Counter usage

Both wrappers and delegates offer a way to access the wrapper/delegate itself. In fact, while we want to access the CRDT through the simple property interface in most of the application codebase, it is also crucial to have access to the CRDT itself to perform native operations such as merges and equalities.

By adopting the syntax and structures explained in this section, we can have a significant advantage: reducing the portion of the codebase that must

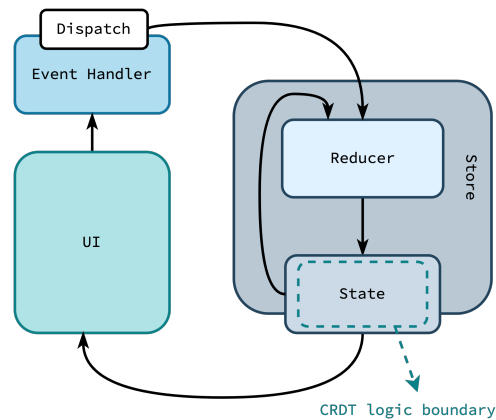


Figure 3.4: Reduced CRDT boundary with Redux pattern

be aware of the CRDT nature of the state. In figure 3.4, we can see how the boundary of the CRDT logic is now only contained in the state.

From this point on, we will refer to properties that encapsulate the CRDT logic using a wrapper/delegate as “wrapped”.

3.3.3 A complete flow

Now that we have introduced both the CRDT hierarchy and the interface we want to use, it is time to give a complete view of the built system and how to use it. Listing 3.6 displays the note-taking application state rewritten using the property wrapper interface.

```

struct AppState {
  @ReplicableDictionary var notes: [UUID: Note]
  var selectedCategories: Set<Category>
  @OptionalReplicable var user: User?

  struct Note {
    @ReplicableRegister var title: String
    @ReplicableRegister var description: String
    @ReplicableRegister var categories: Category
    @ReplicableFlag var pinned: Bool
  }

  struct User {
    @ReplicableRegister var name: String
  }
}
  
```

```

    @ReplicableRegister var age: Int
  }
}

```

Listing 3.6: AppState with property wrappers

As introduced in section 3.3.1, we can look at this state as a hierarchical structure able to handle replicable behavior.

However, what do we mean by replicable behavior? Of course, we imply that all replicable properties should act as such, but we also want the whole state to behave as a composition of those properties. In particular, there are a few methods that we want to implement globally. First and foremost, we want to merge one state with the other. Secondly, we want to compare states, and finally, we may want to add some additional methods such as garbage collection that act on the whole state.

To add all these functionalities, it is necessary to specify one piece of information for each structure: the properties that should be considered replicable. Once this is done, it is possible to programmatically define methods that can iteratively and recursively traverse the tree and evaluate the function on each "node".

Merge

Merging is the strategy that state-based CRDTs use to combine two states. Merge functions are defined to be commutative, associative, and idempotent.

- **CRDTs:** each CRDT implements its merge method. For instance, the Counter CRDT merge functions apply a union of the increments and decrements sets.
- **Replicable structures:** replicable structures merge by merging each of their replicable properties, while leaving non-replicable properties unaltered.
- **Option types:** option types merge by merging their register if at least one value is `null` and by merging their values otherwise.

Equality

It is helpful to define two types of equality: *shallow* and *deep* equality. Shallow equality compares wrapped values, while deep equality compares real values (see Section 2.4.1).

- **CRDTs:** each CRDT implements its equality functions. Deep equality compares all metadata contained in the CRDT; therefore, two CRDT must be identical to conform to this standard of equality. Shallow equality only compares wrapped values.
- **Replicable structures:** replicable structures are equal in depth if all their replicable properties are equal in depth. Therefore, non-replicable properties are not considered in deep equality. Shallow equality takes into consideration all properties of a given structure.
- **Option types:** two option types are equal in-depth, either if they are both null and their replicable register is equal in-depth, or if they have equal in-depth values. Shallow equality only compares the register wrapped values.

Additional methods

Additional methods such as garbage collection are propagated down the tree similarly to what is done with merging and equality.

3.3.4 Composition of maps CRDTs

One last contribution coming from this thesis is the idea of composing maps CRDTs starting from simpler CRDTs to allow for additional flexibility of the data types. The fact that we can trace maps' concurrency semantics back to the semantics of a set and another CRDT is not a coincidence; in fact, maps can be implemented by putting together a set CRDT and a map of CRDTs.

- The set CRDT decides which keys have a corresponding value in the map. We can choose from any set CRDT design to specify how we want the map to behave.
- The map CRDT decides the value associated with each key contained in the set. Therefore all keys present in the set have a one-to-one mapping to the values present in the set.
 - A map is *simple* if its entries are literals. In this case, the internal map will contain Registers.
 - A map is *complex* if its entries are CRDTs themselves. In this case, the internal map will contain any other CRDT design.

The concept of associating the map's keys to a Set specification and the entries to a CRDT one is not an innovation (see Section 2.5). Nevertheless, a new map should have been defined for each CRDT specification in previous implementations because each CRDT interface was different and required a slightly different algorithm. By remodeling the CRDT interface by offering a single and general update method, we can collapse all these implementations together.

Editing the map

In listing 3.7 the algorithm for setting a value in an entry of the map is shown.

```
switch (oldValue, newValue) {
  // insert
  case (.none, .some(let newValue)):
    self.replicableMap[key] = Entry(wrappedValue: newValue)
    self.replicableSet.insert(key)
  // remove
  case (.some, .none):
    self.replicableMap[key] = nil
    self.replicableSet.remove(key)
  // update
  case (.some(let oldValue), .some(let newValue))
    where oldValue != newValue:
    self.replicableMap[key]?.wrappedValue = newValue
    self.replicableSet.updateEntry(key)
  default:
    return
}
```

Listing 3.7: Composed dictionary behaviour

- **Insert:** we insert the key in the set and insert a new CRDT with a specific wrapped value in the map.
- **Remove:** we remove the key from the set and remove the value associated with that key in the map.
- **Update:** we set the new wrapped value in the map and update the set. In this case, the `updateEntry` function is called directly on the CRDT; this is in order to allow for additional flexibility. In fact, from the set

point of view, we may want to consider updates similarly as insertions or choose to ignore them.

Merge

Merging two maps is equivalent to merging the related set and map. In listing 3.8 we can have a look at the algorithm. First, we merge their sets, and second, we use the resulting set's wrapped value to fill an empty map with entries. If only one replica contains the entry, that entry is inserted directly; if both replicas contain the entry, a recursive merge is performed.

```
result.replicableSet = self.replicableSet
                        .merged(with: other.replicableSet)

for key in result.replicableSet.wrappedValue {
  switch (self.replicableMap[key], other.replicableMap[key]) {
    case (.some(let element), .none), (.none, .some(let element)):
      result.replicableMap[key] = element
    case (.some(let this), .some(let other)):
      result.replicableMap[key] = this.merged(with: other)
  }
}
```

Listing 3.8: Composed dictionary merge

Chapter 4

Results and Analysis

This chapter describes the strategy used to evaluate the thesis, presents the results, and discusses them. The chapter is divided into two sections which refer respectively to the sections defined in Chapter 3.

4.1 Client-Server architecture

4.1.1 Evaluation strategy

This section aims to justify quantitatively the design choices explained in the methodology section (see Section 3.2). More in detail, the following choices will be evaluated:

- Adopting state-based CRDTs over operation-based ones
- Quantitative evaluation of the advantage of sporadically performing garbage collection
- Evaluation of the number of merges in the client-server architecture compared to the peer-to-peer one

4.1.2 Benchmark on CRDT types

We have motivated the design choice of adopting state-based CRDTs by stating that it leads to a reduction in the size of the occupied memory of the server (see Section 3.2.1). This statement comes from the idea that even if operations are smaller than the whole state, they have to be kept in memory as long as all devices read them. On the other hand, if we save the state, we are interested only in the newest value.

To properly evaluate this, we should perform a few benchmarks comparing state-based and operation-based designs' memory occupation. Nevertheless, this requires fully implementing operation-based CRDTs, which falls outside the scope of the thesis. Therefore, we only present a single benchmark run on an LWWSet (see Section 2.5.4) that can help us visualize the CRDT behavior and understand the reasoning behind the statement.

This benchmark compares a state-based and operation-based version of an LWWSet under the assumption that twenty insert operations are performed for ten different elements. This benchmark also supposes that the replicas synchronize with the server after each update and that no garbage collection is performed.

Results and Discussion

Figure 4.1 shows the results of the benchmark. The size reported on the ordinal axis is the size of the serialized CRDT, which corresponds to the format on the server. During the first ten operations, distinct elements are inserted; from the eleventh operation, the benchmark adds elements already present in the set. We can see in the figure that the state-based solution size is slightly larger than the operation-based one during the first ten operations. We can also notice how the operation-based implementation surpasses the state-based one from the eleventh operation onwards. This behavior can be explained by the fact that when adding elements already present in the set, the state-based solution only needs to update the timestamp associated with each element; on the other hand, the operation-based solution keeps storing messages of the same size.

4.1.3 Benchmark on garbage collection

Garbage collection can be implemented by adding small doses of synchronization to perform a global reset (see Section 2.4.3). In this section, we want to evaluate the effect that garbage collection of all tombstones can have on the size of the app state. We take the state of the mock note-taking application as a case study. We evaluate the space saved by garbage collection in different phases of mock application execution. The mock execution is made up of these steps:

1. Login
2. Add twenty notes
3. Edit twenty notes (modify title and description, add a category)



Figure 4.1: Memory occupation comparison between state and operation based LWWSet

4. Remove the same twenty notes
5. Logout

It must be highlighted that garbage collection can only be performed if we are forcing strong synchronization or if we are sure that all replicas are up to date.

Results and Discussion

Figure 4.2 shows the size of the application state after performing each modification. In the first part of the graph, the trend is monotonically increasing for both solutions, and we can start noticing a difference between the two lines from the “RemoveNotes” operation. While previous operations only added actual data, this is the first operation transforming data into tombstones. The solution with garbage collection goes back to the same size it had before adding any note. On the other side, the one without garbage collection maintains approximately 4000 Bytes of tombstones. At the end of the execution, the solution without garbage collection is approximately three times as big as the one with garbage collection.

From this benchmark we can see that the impact tombstones have on the state after just a few operations is significant. This observation justifies the idea of sporadically performing a complete garbage collection on the state.

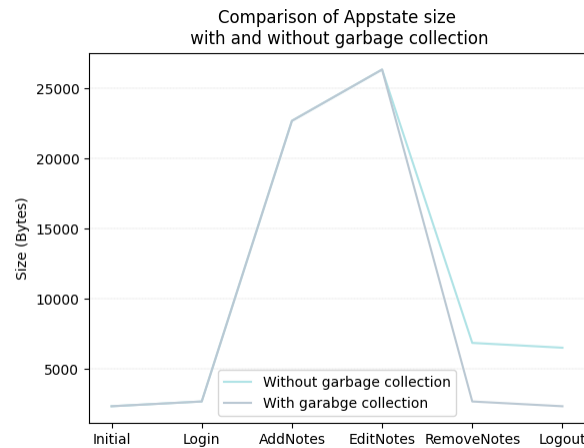


Figure 4.2: Comparison of Appstate size with and without garbage collection

4.1.4 Number of merges

This section evaluates the variation in the number of merges given a peer-to-peer or client-server architecture. This value changes depending on the synchronization strategy that we adopt (see Section 3.2.3). Therefore, instead of evaluating the number of merges over time, we choose to evaluate them in a scenario where n replicas are trying to synchronize their local changes. Given this scenario, we want both architectures to reach a final state where all replicas have a consistent and complete state. Within the client-server architecture, we will need each replica to synchronize with the server, performing one merge. On the other hand, within the peer-to-peer architecture, we will need each replica to send its state to the other $n - 1$ replicas, which will perform one merge each.

Therefore in this scenario, complete synchronization with a client-server architecture requires $n - 1$ merges (assuming the server as initially empty), while in the peer-to-peer architecture, it requires $n * (n - 1)$ merges.

4.2 Integration into code base

4.2.1 Evaluation strategy

This section aims to evaluate the integration into the codebase. To achieve this goal, it considers three scenarios and compares them from a quantitative point of view on different dimensions.

The three scenarios are the following:

1. **Base:** a scenario that does not use CRDTs.
2. **Replicable:** a scenario that uses CRDTs directly, by declaring them explicitly. It also uses the traditional queries and updates functions as described in the literature.
3. **Wrapped:** a scenario that uses CRDTs indirectly: it declares them implicitly (making use of property wrappers). This scenario uses the `getWrappedValue`, `setWrappedValue` methods (see Section 3.3.2).

The first part of the evaluation will be conducted through some benchmarks. The benchmarks will evaluate the change in execution time and will be divided into two sections: the first will be more fine-grained and benchmark CRDTs individually. On the other side, the second will try to give a complete picture by benchmarking the influence of their use in the note-taking application. The second part will evaluate the impact of the scenario choice regarding the portability of code. This goal will be reached by examining the lines of code (LOC) necessary to implement the note-taking application in the three scenarios.

4.2.2 Benchmarks on CRDT designs

This section of the benchmarks considers CRDT designs individually. For each design it compares the execution time for the following operations:

- Initialization
- Query method (read)
- Update methods

While the initialization benchmarks and the query method are present for each design, the update methods vary in number and type depending on the CRDT. The initialization and query method execution time will be approximately equal in the replicable and wrapped scenario for all designs, because this part remains identical in the two scenarios.

Benchmark visualization

For each benchmark two elements are shown:

- A graph showing mean and standard deviation of the execution time for each operation in the three scenarios. This graph was chosen to give a general view of how CRDT based solutions perform compared to the base scenario.
- A table showing the percent variation of each update operation from the replicable to the wrapped scenario ($\frac{mean_{wrapped} - mean_{replicable}}{mean_{replicable}} \times 100$). In this case, the initialization and query operations are not considered, given that their value is approximatively equal in all designs.

CRDT designs

The following CRDTs are examined in this section:

- Register
 - Register<Int>
- Counter
 - GCounter<Int>
 - PNCounter<Int>
- Counter
 - AddOnlySet<Int>
 - LWSet<Int>
 - PNSet<Int>
- Map
 - Map<LWSet<Int>, Register<Int>>
 - Map<LWSet<Int>, Counter<Int>>
- List/Sequence
 - Array<Int>

Testing enviroment

Tests have been run on MacBook Pro (2019) with processor 2,4 GHz 8-Core Intel Core i9. Each benchmark has been run 1.000.000 times, with 10.000 warm-up iterations with Google's swift-benchmark library [42]. The IQR method has been used to identify outliers, which have been not considered in the evaluation.

Results

Register The register benchmark shown in graph 4.3 was performed on three operations: initialization, read and set.

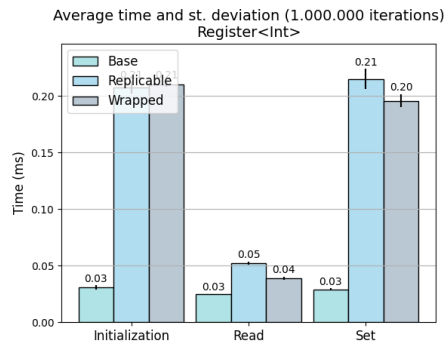


Figure 4.3: Register's benchmarks

Table 4.1 shows that operations from the wrapped scenario have a similar execution time compared to the ones of the replicable one. This similarity can be explained because Register CRDTs have just one update function, which is logically equivalent to setting the wrapped value.

Design	Set (%)
Register<Int>	-9.00

Table 4.1: Register's operations percent variation from replicable to wrapped scenario

Counter In figure 4.4 we can see the execution time of the Grow-only Counter put in comparison to the Positive-Negative Counter.

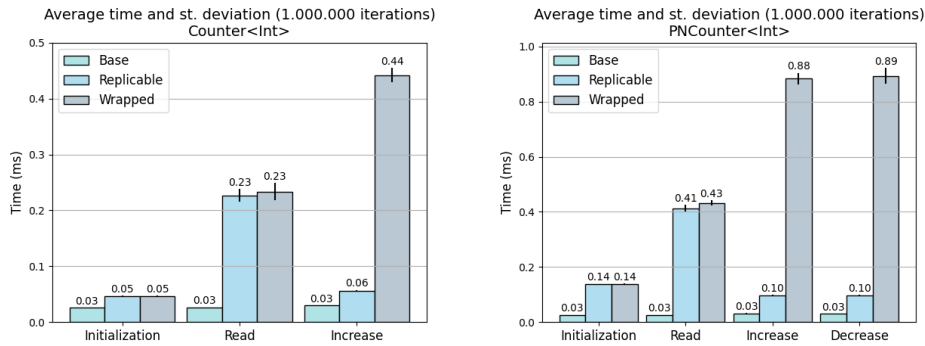


Figure 4.4: Counters' benchmarks

Table 4.2 shows that both counters behave similarly regarding the variation of execution time from the replicable to the wrapped scenario. In both cases, the increase operation leads to a time increase around the 700-800% mark. The same also holds for the decrease operation of the PNCOUNTER (which is not a valid operation for the GCounter).

Design	Decrease (%)	Increase (%)
Counter<Int>	nan	691.41
PNCOUNTER<Int>	823.14	813.47

Table 4.2: Counters's operations percent variation from replicable to wrapped scenario.

Set In figure 4.5 three sets designs are taken into consideration: AddOnlySet, LWWSet and PNSet. Depending on how the logic of each design is implemented, we can see how the update operations' execution time differs significantly. If we put in comparison the LWWSet and PNSet with the AddOnlySet, we can see a big difference in the execution time for the insert operation. This difference is also a consequence of the complexity of the read operation: we must keep in mind that all wrapped CRDTs must calculate this value to decide which operations to apply.

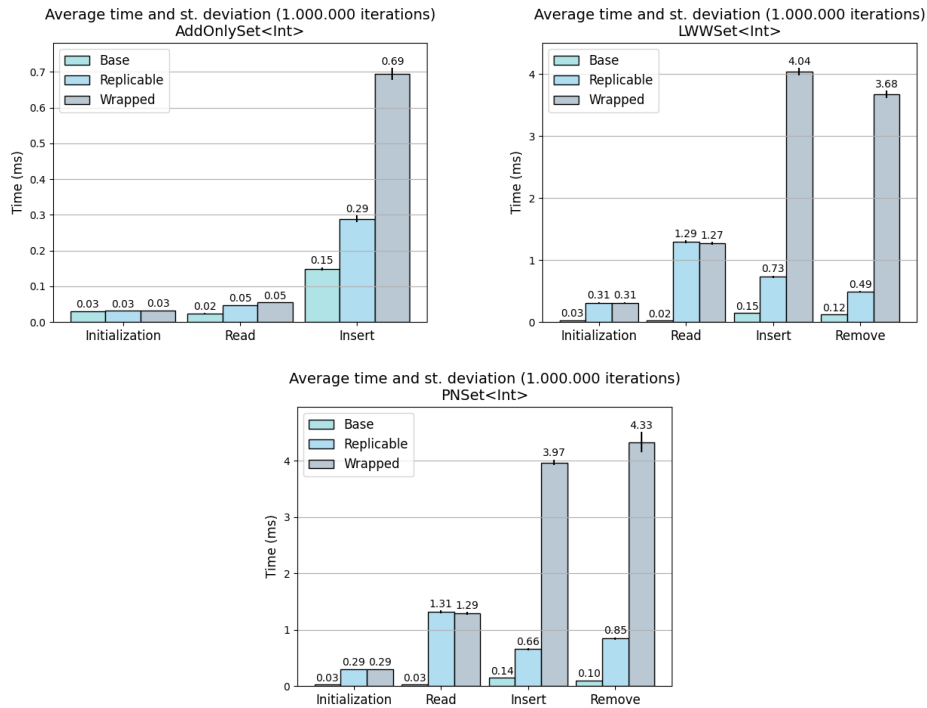


Figure 4.5: Sets' benchmarks

Table 4.3 also highlights the fact that choosing between different designs can have a significant impact on update operations. This phenomenon is particularly evident for the insert operation, where percent variations can lead from 100% up to 500% depending on the design.

Design	Insert (%)	Remove (%)
AddOnlySet<Int>	140.02	nan
LWWSet<Int>	451.56	652.46
PNSet<Int>	505.45	411.71

Table 4.3: Sets's operations percent variation from replicable to wrapped scenario.

Map Maps are a composition of a Set and another CRDT (see Section 3.3.4). In the benchmark shown in figure 4.6 we put in comparison four maps obtained by combining a LWWSets with a Counter or Register CRDT. The first can fall under the category of map of CRDT, while the second is a map of literals.

Generally, we expect execution times to be influenced by the Set and CRDT

design that compose the map. In fact, in figure 4.6 we can see how the Register map has a better performance than the Counter map.

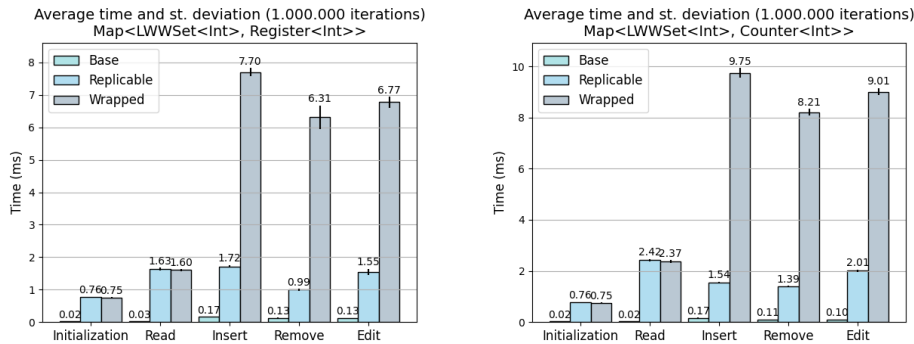


Figure 4.6: Maps' benchmarks

In table 4.4 we do not see a big difference between the two maps. The overhead is given mainly by the map logic that checks whether each element has been added, removed, or updated, which is equally present in both benchmarks.

Design	Edit (%)	Insert (%)	Remove (%)
Map<LWWSet<Int>, Counter<Int>>	348.09	532.39	489.64
Map<LWWSet<Int>, Register<Int>>	336.62	348.72	534.07

Table 4.4: Maps's operations percent variation from replicable to wrapped scenario.

List/Sequence Calculating the operations that transform an array from an initial to a final state is complex (see Section 3.3.2). In figure 4.7 we can observe how this factor greatly influences the execution time of the wrapped scenario.

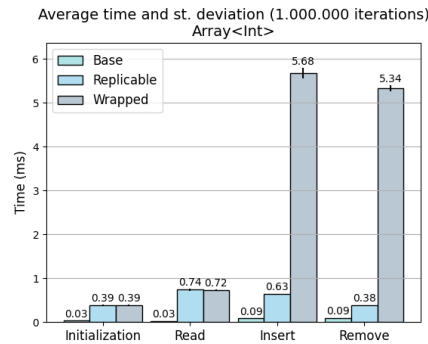


Figure 4.7: Array's benchmark

In table 4.5 we can see how the Array has the highest percent variation from replicable to wrapped scenario both for the insert and remove operations among all studied CRDT designs.

Design	Insert (%)	Remove (%)
Array<Int>	795.07	1316.85

Table 4.5: Array's operations percent variation from replicable to wrapped scenario.

4.2.3 Benchmarks on note-taking application

The second section of benchmarks considers the note-taking app example and takes into consideration the execution time of a mock application run.

The mock run is made up of these steps:

1. Login
2. Add n notes
3. Edit n notes (modify title, description and categories)
4. Set n notes as pending
5. Remove n notes
6. Logout

This benchmark aims to give a broader understanding of how using one of the two CRDT solutions influences the application's execution time. Naturally, this benchmark is tightly dependent on the application's architecture.

Testing environment

Tests have been run on an Iphone 12. Each benchmark has been run 1.000 times.

Results

As expected, the results displayed in figure 4.8 show again that the replicable scenario performs better than the wrapped one because of the additional computational complexity of the wrapped CRDTs. Nevertheless, the relative difference between the average execution values is not huge between the three scenarios: the base scenario average value falls around 50 ms, the replicable scenario falls around 57 ms, and the wrapped scenario falls around 85 ms. When comparing them with the base scenario, the replicable one averages a 26% increase in execution time, while the wrapped averages a 68% increase. Finally, we calculate, similarly to what we have done with CRDT designs, the percent variation from the replicable to the wrapped scenario, which results in 33%.

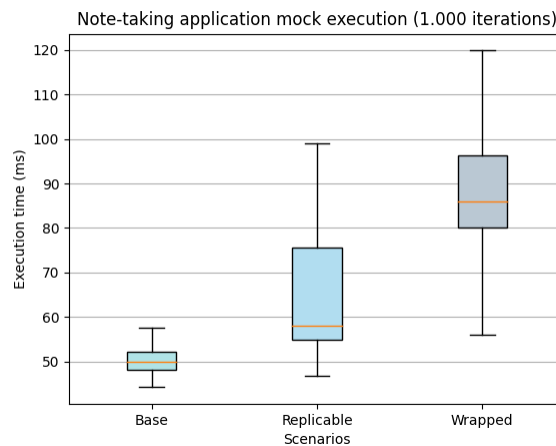


Figure 4.8: Note-taking application mock execution benchmark

4.2.4 Lines of code comparison

This last evaluation step aims to measure the code portability in the different scenarios quantitatively, namely how flexibly we can switch from a non-CRDT solution to a CRDT one. Therefore, we consider the LOCs necessary to implement the different scenarios. In particular, we look at what these LOCs

implement within the Redux pattern. It is noteworthy to highlight that these numbers do not comprehend the lines of code contained in the CRDT library.

Results

The Venn diagram in figure 4.9 shows how the lines of code of the whole application are distributed between the different scenarios. First, we can see that most of the code is shared within all implementations. Secondly, we can notice an increase in the line of codes in both solutions implementing CRDTs. Nevertheless, the most relevant aspect is the size of this increase in the replicable and wrapped scenario. As a matter of fact, the replicable scenario requires approximately two times as many lines of code to implement the same feature.

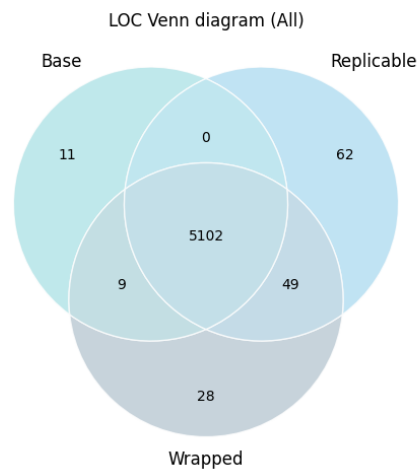


Figure 4.9: Note-taking application line of codes distribution (whole application)

If we want to understand more in detail how the wrapped scenario entails a reduction in LOCs, we can look at the two diagrams in figure 4.10. The diagram on the left shows the LOC distribution for the Redux store, while the one on the right shows the LOC distribution for the reducers. We can see how these two design pattern elements contain all the lines of code present exclusively in the replicable and wrapped implementations (62 for the replicable scenario and 28 for the wrapped one).

In the replicable scenario, the 62 lines are divided between the Redux application state and the reducers. The store properties must be declared as

CRDT types, and the reducers must be rewritten to modify the CRDT types instead of the basic ones.

While the store properties of the wrapped scenario must be declared slightly differently from the base one, the reducers do not need modification. Therefore, the 28 lines are only due to the Redux store definition in this case.

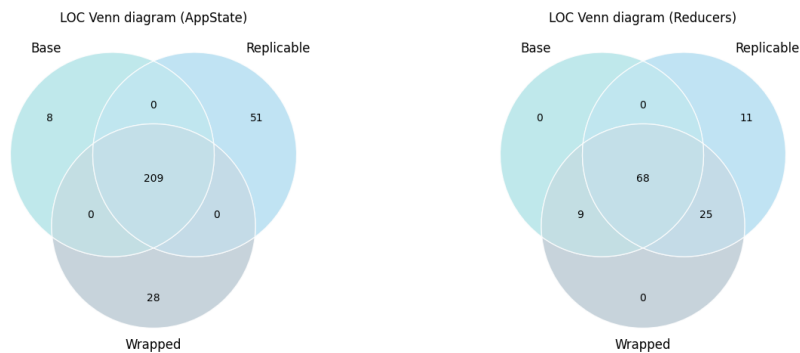


Figure 4.10: Note-taking application lines of code distribution (AppState and Reducers)

4.2.5 Discussion

Both the first and second sections of benchmarks show that the wrapped solution performs worse than the replicable solution from an execution time point of view. This phenomenon is due to the overhead given by interpreting which operations are necessary for each CRDT to move from the initial state to the final state. This overhead is more meaningful in some designs and almost insignificant in others (i.e., the Register CRDT), reaching up to a 1300% of execution time increase for complex data structures such as the arrays. On the other side, the benchmark performed on the mock execution of the note-taking application shows that the wrapped scenario had a 33% increase in execution time compared to the replicable one. This data indicates that by putting the CRDTs into their context of use, the actual overhead does not influence execution time as drastically. On the other side, the information regarding the lines of code clearly shows that the wrapped scenario requires fewer code modifications and is, in general, easier to handle since it does not require editing the logic contained in the Redux reducers.

Therefore, it is clear that deciding how to add a CRDT feature depends on striking a compromise between performance and code portability.

Chapter 5

Conclusions and Future work

5.1 Conclusions

The first objective of this thesis was to discuss the design choices that must be taken given a client-server architecture and the adoption of Conflict-free Replicated Data Types. We determined that state-based CRDTs are the best choice because they lead to decreased memory occupation given the dynamic environment. We supported this statement with a benchmark. Furthermore, we stated that Hybrid Logical clocks are the optimal logical clock in the given scenario, and we can pair them with Vector Clocks if the casual order of events is required. We also discussed which strategies we can adopt to perform synchronization and what having a system composed of dynamic nodes entails.

The second objective was to investigate a method to build convergent mobile application states and evaluate its advantages and disadvantages. We identified the elements that compose the application state, their hierarchy, and how they can be composed together. Furthermore, we proposed remodeling the CRDT interface, making the CRDT feature easier to implement. We have also explained how the new interface allows for a more flexible map CRDT implementation. We have confirmed that the proposed remodeling leads to a decrease in performance. However, we have shown that the increase in execution time may fall in an acceptable range given the advantage of implementing portable code.

All in all, we can say that we have been able to answer the research questions proposed by this thesis thoroughly, and we have offered a valid solution to integrate CRDTs seamlessly within a mobile stack.

5.2 Limitations

The main limitations in this thesis are related to the results. Concerning the client-server architecture, the proposed benchmarks give the reader a basic understanding of the reasons that support the design choices. Nevertheless, these benchmarks are incredibly simplified and are not meant to be considered actual proof of the statements. Concerning the integration into the codebase, the comparison of the lines of code is relevant because it gives us a high-level understanding of how opting for one solution or the other influences the codebase's structure and the developers' work. Nevertheless, this evaluation has several drawbacks. Firstly, it is done on a simplified application and therefore cannot exhaustively represent what would happen in a real application scenario. Secondly, it operationalizes the code portability by examining the number of lines of codes and their position in the codebase; this may be a relevant data point, but a decrease in the number of lines of code does not directly imply a more maintainable codebase. Therefore, we should also evaluate the experience of the developers adopting the different solutions.

5.3 Future Work

Future work has to focus primarily on a more complete evaluation of what has been proposed in this thesis. Concerning the client-server architecture, it would be interesting to conduct more generalized benchmarks about the server's memory occupation and the outcomes of garbage collection. Concerning the integration into the codebase, it would be interesting to study whether users would perceive the execution time increase and how it would influence their user experience. On the other side, it would also be valuable to evaluate the experience of developers adding the CRDT feature in the two ways described by the thesis.

Furthermore, it would be relevant to evaluate this solution in production. Therefore, we hope to have the opportunity to apply the library developed in parallel to the thesis to one of the products of Bending Spoons.

5.4 Reflections

The work performed in this thesis does not relate to any relevant ethical aspects: no experiment involving people has been conducted, and the thesis does not contain any private or sensitive data. The thesis does not strictly relate

to sustainability either; nevertheless, we could see the considerations about performance and synchronization from the energy consumption point of view. In general, decreasing the synchronization cost could lead to a decrease in the energetic impact of the system.

References

- [1] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free Replicated Data Types,” in *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds., ser. Lecture Notes in Computer Science, vol. 6976, Grenoble, France: Springer, Oct. 2011, pp. 386–400. DOI: [10.1007/978-3-642-24550-3_29](https://hal.inria.fr/hal-00932836). [Online]. Available: <https://hal.inria.fr/hal-00932836>.
- [2] grrowl, *Grrowl/redux-scuttlebutt*. [Online]. Available: <https://github.com/grrowl/redux-scuttlebutt>.
- [3] M. Kleppmann and A. R. Beresford, “A conflict-free replicated json datatype,” *IEEE Transactions on Parallel & Distributed Systems*, vol. 28, no. 10, pp. 2733–2746, Oct. 2017, ISSN: 1558-2183. DOI: [10.1109/TPDS.2017.2697382](https://doi.org/10.1109/TPDS.2017.2697382).
- [4] [Online]. Available: <https://bendingspoons.com/>.
- [5] *Desktop vs mobile vs tablet market share worldwide*. [Online]. Available: <https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide#yearly-2011-2021>.
- [6] Y. Wurmser, *Apps far outpace browsers in us adults’ mobile time spent*, Jul. 2020. [Online]. Available: <https://www.emarketer.com/content/the-majority-of-americans-mobile-time-spent-takes-place-in-apps>.
- [7] Google, *Eric schmidt at mobile world congress*, Mar. 2010. [Online]. Available: https://www.youtube.com/watch?v=ClkQA2Lb_iE.
- [8] C. Baquero and F. Moura, “Specification of convergent abstract data types for autonomous mobile computing,” Departamento de Informática, Universidade do Minho, Tech. Rep., 1997.

- [9] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, “Local-first software: You own your data, in spite of the cloud,” in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2019, Athens, Greece: Association for Computing Machinery, 2019, pp. 154–178, ISBN: 9781450369954. DOI: [10.1145/3359591.3359737](https://doi.org/10.1145/3359591.3359737). [Online]. Available: <https://doi.org/10.1145/3359591.3359737>.
- [10] E. Brewer, “Towards robust distributed systems,” Jan. 2000, p. 7. DOI: [10.1145/343477.343502](https://doi.org/10.1145/343477.343502).
- [11] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002, ISSN: 0163-5700. DOI: [10.1145/64585.564601](https://doi.org/10.1145/64585.564601). [Online]. Available: <https://doi.org/10.1145/64585.564601>.
- [12] Y. Saito and M. Shapiro, “Optimistic replication,” Tech. Rep. MSR-TR-2003-60, Oct. 2003, p. 50.
- [13] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford, “Verifying strong eventual consistency in distributed systems,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. DOI: [10.1145/3133933](https://doi.org/10.1145/3133933). [Online]. Available: <https://doi.org/10.1145/3133933>.
- [14] P. Bailis and A. Ghodsi, “Eventual consistency today: Limitations, extensions, and beyond,” *Commun. ACM*, vol. 56, no. 5, pp. 55–63, May 2013, ISSN: 0001-0782. DOI: [10.1145/2447976.2447992](https://doi.org/10.1145/2447976.2447992). [Online]. Available: <https://doi.org/10.1145/2447976.2447992>.
- [15] N. Preguiça, *Conflict-free replicated data types: An overview*, 2018. arXiv: [1806.10254](https://arxiv.org/abs/1806.10254) [cs.DC].
- [16] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” vol. 21, no. 7, pp. 558–565, Jul. 1978, ISSN: 0001-0782. DOI: [10.1145/359545.359563](https://doi.org/10.1145/359545.359563). [Online]. Available: <https://doi.org/10.1145/359545.359563>.
- [17] F. Mattern, “Virtual time and global states of distributed systems,” in *PARALLEL AND DISTRIBUTED ALGORITHMS*, North-Holland, 1988, pp. 215–226.

- [18] C. J. Fidge, “Timestamps in message-passing systems that preserve the partial ordering,” in *Proc. 11th Austral. Comput. Sci. Conf. (ACSC '88)*, 1988, pp. 56–66.
- [19] M. Demirbas, M. Leone, B. Avva, D. Madeppa, and S. S. Kulkarni, “Logical physical clocks and consistent snapshots in globally distributed databases,” 2014.
- [20] D. Murat, *Hybrid logical clocks*, Oct. 2014. [Online]. Available: <http://muratbuffalo.blogspot.com/2014/07/hybrid-logical-clocks.html>.
- [21] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “A comprehensive study of Convergent and Commutative Replicated Data Types,” Inria – Centre Paris-Rocquencourt ; INRIA, Research Report RR-7506, Jan. 2011, p. 50. [Online]. Available: <https://hal.inria.fr/inria-00555588>.
- [22] C. Baquero, P. S. Almeida, and A. Shoker, “Making operation-based crdts operation-based,” in *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, ser. PaPEC '14, Amsterdam, The Netherlands: Association for Computing Machinery, 2014, ISBN: 9781450327169. DOI: [10 . 1145 / 2596631 . 2596632](https://doi.org/10.1145/2596631.2596632). [Online]. Available: <https://doi.org/10.1145/2596631.2596632>.
- [23] P. Almeida, A. Shoker, and C. Baquero, “Efficient state-based crdts by delta-mutation,” Oct. 2014.
- [24] C. Baquero, P. S. Almeida, and A. Shoker, “Pure operation-based replicated data types,” *CoRR*, vol. abs/1710.04469, 2017. arXiv: [1710 . 04469](https://arxiv.org/abs/1710.04469). [Online]. Available: <http://arxiv.org/abs/1710.04469>.
- [25] 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*, ser. VMIL 2019, Athens, Greece: Association for Computing Machinery, 2019, pp. 48–57, ISBN: 9781450369879. DOI: [10 . 1145 / 3358504 . 3361231](https://doi.org/10.1145/3358504.3361231). [Online]. Available: <https://doi.org/10.1145/3358504.3361231>.
- [26] S. Weiss, P. Urso, and P. Molli, “Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks,” in *2009 29th IEEE International Conference on Distributed Computing Systems*, 2009, pp. 404–412. DOI: [10 . 1109 / ICDCS . 2009 . 75](https://doi.org/10.1109/ICDCS.2009.75).

- [27] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, “Replicated abstract data types: Building blocks for collaborative applications,” *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 354–368, 2011, ISSN: 0743-7315. DOI: <https://doi.org/10.1016/j.jpdc.2010.12.006>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0743731510002716>.
- [28] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, “A commutative replicated data type for cooperative editing,” in *2009 29th IEEE International Conference on Distributed Computing Systems*, 2009, pp. 395–403. DOI: [10.1109/ICDCS.2009.20](https://doi.org/10.1109/ICDCS.2009.20).
- [29] V. B. d. Sousa, *Key-crdt stores*, 2012. [Online]. Available: <http://hdl.handle.net/10362/7802>.
- [30] B. Sypytkowski, *State-based crdts: Maps*. [Online]. Available: <https://bartoszsypytkowski.com/crdt-map/>.
- [31] *Redux - a predictable state container for javascript apps.: Redux*. [Online]. Available: <https://redux.js.org/>.
- [32] BendingSpoons, *Bendingspoons/katana-swift*. [Online]. Available: <https://github.com/BendingSpoons/katana-swift>.
- [33] automerge, *Automerge/automerge*. [Online]. Available: <https://github.com/automerge/automerge#readme>.
- [34] Yjs, *Yjs/yjs: Shared data types for building collaborative software*. [Online]. Available: <https://github.com/yjs/yjs>.
- [35] [Online]. Available: <https://riak.com/>.
- [36] [Online]. Available: <https://www.ditto.live/>.
- [37] Qiantan hong / *crdt.el*. [Online]. Available: <https://code.librehq.com/qhong/crdt.el>.
- [38] E. Wallace, *How figma’s multiplayer technology works*. [Online]. Available: <https://www.figma.com/blog/how-figmas-multiplayer-technology-works/>.
- [39] E. W. Myers, “An o(nd) difference algorithm and its variations,” *Algorithmica*, vol. 1, pp. 251–266, 1986.
- [40] [Online]. Available: <https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID617>.
- [41] [Online]. Available: <https://kotlinlang.org/docs/delegated-properties.html>.

- [42] [Online]. Available: <https://github.com/google/swift-benchmark>.

