



VRIJE
UNIVERSITEIT
BRUSSEL



Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Applied Sciences and Engineering:
Computer Science

CSCRIPT: A DISTRIBUTED PROGRAMMING LANGUAGE FOR AVAILABLE AND CONSISTENT SHARING OF OBJECTS

Kevin De Porre

June 2018

Promotor:
Prof. Dr. Elisa Gonzalez Boix

Advisors:
Florian Myter
Christophe De Troyer

Sciences and Bio-Engineering Sciences

Abstract

For decades developers of distributed systems assumed a stable connection between clients and servers. Recently, we witnessed an increase in hardware innovations leading to a vast collection of Internet-connected devices, such as smartphones and smartwatches. As these devices become ever more portable user mobility increases leading to volatile connections. This forces developers to design applications that are robust against partial failures. In other words, applications should provide functionality that is available even offline.

The CAP theorem (Brewer, 2000) states that distributed systems cannot remain both available and consistent under network partitions. This forces programmers to choose between availability and consistency. Many languages provide support to encode available data, e.g. by copying objects across the network. However, programmers need to manually implement a consistency layer to keep the data synchronized. This task is error-prone given the inherent complexity of consistency models. Recently, replicated data types were proposed to alleviate these issues (Shapiro, Preguiça, Baquero, & Zawirski, 2011b). However, only a limited portfolio of data types are available (e.g. lists and sets). So far, there is no generally applicable approach to designing available systems. As a result, programmers implement ad hoc solutions which require advanced knowledge of replication and consistency.

In this thesis we propose CScript, a distributed programming language providing two types of distributed objects: available and consistent objects. Applications can share these objects in order to achieve high availability while keeping critical parts consistent. The novelty of the language are its high-level constructs for availability and consistency, which free the programmer from low-level concerns such as replication and data consistency.

CScript introduces SECROs, a novel general-purpose replicated data type. Programmers use dedicated language constructs, called state validators, to declare the behaviour of SECROs in the face of concurrent operations. This information is used to automatically detect and solve conflicts. As such, SECROs form a general-purpose solution to designing available systems, thereby omitting the need for manual ad hoc approaches.

To evaluate our solution we compare a real-time collaborative text editor built atop SECROs with an implementation that uses a state-of-the-art available data type, namely JSON CRDTs (Kleppmann & Beresford, 2017). The evaluation consists of a qualitative and a quantitative analysis. The former compares the text editors from a code viewpoint. The latter performs a number of benchmarks that quantify various properties of both approaches. Our results show that SECROs are more flexible than the traditional CRDT approach. The benchmarks also show that SECROs efficiently manage memory but incur a performance overhead.

Samenvatting

Jarenlang zijn programmeurs van gedistribueerde systemen uitgegaan van stabiele verbindingen tussen de gebruikers en de servers. Onlangs waren we getuige van een toename in hardware innovaties, leidende tot een waaier aan nieuwe toestellen die verbonden zijn met het Internet, zoals smartphones en smartwatches. Aangezien deze toestellen draagbaarder worden zijn de gebruikers mobieler, hetgeen leidt tot onstabiele verbindingen. Deze onstabiele vereist dat programmeurs applicaties bouwen die robust zijn tegen gedeeltelijke storingen van het gedistribueerde systeem. Applicaties moeten dus offline functionaliteit aanbieden, in de mate van het mogelijke.

Het CAP theorema (Brewer, 2000) stelt dat gedistribueerde systemen niet beschikbaar én consistent kunnen zijn wanneer netwerk partities zich voordoen. Dit dwingt de programmeur om te kiezen tussen beschikbaarheid en consistentie. Ook al dienen de meeste programmeurs deze keuze te maken zijn er geen gedistribueerde programmeertalen die hulp aanbieden voor de ontwikkeling van zowel beschikbare als consistente systemen. Vele talen bieden manieren aan om data beschikbaar te maken, bv. door objecten te kopiëren over het netwerk. Programmeurs dienen echter manueel een consistentie laag te implementeren die deze data consistent houdt. Dit is een moeilijke opdracht aangezien de complexiteit van consistentie modellen.

Onlangs werden gerepliceerde datatypes voorgesteld om deze problemen tegemoet te komen (Shapiro et al., 2011b). Er is echter maar een beperkt portfolio aan beschikbare datatypes, bv: lijsten en verzamelingen. Tot nu toe is er geen algemeen toepasbare aanpak voor de ontwikkeling van beschikbare systemen. Dit dwingt de programmeur om applicatie specifieke oplossingen te gebruiken, hetgeen gevorderde kennis van replicatie en consistentie vergt.

In dit proefschrift stellen we CScript voor, een gedistribueerde programmeertaal die twee soorten objecten aanbiedt: beschikbare en consistente objecten. Applicaties kunnen objecten delen om hoge beschikbaarheid te behalen maar kritische delen toch consistent te houden. De innovatie van de taal zijn de high-level constructies voor beschikbaarheid en consistentie, welke de programmeur bevrijden van low-level problemen zoals replicatie en data consistentie.

CScript introduceert SECRO's, een nieuw gerepliceerd datatype dat algemeen toepasbaar is. Programmeurs gebruiken specifieke abstracties, genaamd state validators, om het gedrag van SECRO's te definiëren in het geval van gelijktijdige operaties. Deze informatie wordt gebruikt om conflicten automatisch op te sporen en op te lossen. Dit zorgt ervoor dat SECRO's een algemeen toepasbare aanpak zijn voor de ontwikkeling van beschikbare systemen waarbij geen applicatie-specifieke mechanismes nodig zijn.

Om onze oplossing te evalueren vergelijken we een real-time gedeelde

tekst editor die gebouwd is bovenop SECRO's met een versie die gebouwd is bovenop een state-of-the-art beschikbaar datatype, namelijk JSON CRDTs (Kleppmann & Beresford, 2017). De evaluatie bestaat uit twee delen: een kwalitatief en een kwantitatief onderzoek. Het kwalitatief onderzoek vergelijkt de implementatie van beide text editors. Het kwantitatieve onderzoek voert een aantal experimenten uit die verscheidene performantie aspecten van de tekst editors kwantificeren. Onze resultaten tonen aan dat SECRO's expressiever en flexibeler zijn dan de traditionele aanpak. Bovendien onthullen de experimenten dat SECRO's efficiënt omgaan met geheugen maar een aanzienlijke performantie kost met zich meedragen.

Acknowledgements

In the following, I express my gratitude to those who directly or indirectly influenced this dissertation for the better.

First, I would like to thank professor Elisa Gonzalez Boix for the time and efforts she spent in proof-reading this thesis but also for her faith in me and the motivating talks throughout this entire year. Second, a big THANK YOU goes out to my advisors, Florian Myter and Christophe De Troyer, for their continuous feedback on this dissertation but also for our countless discussions and their timely answers to my questions every single day.

In addition to the aforementioned geniuses, there is a special group of beings that need to be thanked, which are my family. I would like to thank my parents for their everlasting love and support. Furthermore, I am very grateful to my grandfather who is always there for me and made me who I am today. Finally, a special sign of gratitude goes out to my wonderful grandmother, who, although no longer with us, continues to inspire me.

How else could I conclude, than by thanking my girlfriend Ellen De Kock for her endless love and support. Ever since we met, she has been there for me. I cannot stress enough how grateful I am to have such a lovely girlfriend. Ik hou van u schatteke!

Declaration of Originality

I hereby declare that this thesis was entirely my own work and that any additional sources of information have been duly cited. I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this thesis has not been submitted for a higher degree to any other University or Institution.

Contents

1	Introduction	
1.1	Research Context	2
1.2	Problem Statement	3
1.3	CScript: An Object-oriented Approach	4
1.4	Contributions	5
1.5	Thesis Structure	5
2	Literature Study	
2.1	Distributed Systems	8
2.2	CAP Theorem	9
2.3	Consistency Models	11
2.3.1	Strong Consistency	12
2.3.2	Eventual Consistency	12
2.3.3	Strong Eventual Consistency	12
2.4	Consistency Protocols	13
2.4.1	Strong Consistency	13
2.4.2	Eventual Consistency	13
2.4.3	Strong Eventual Consistency	14
2.5	Conflict-free Replicated Data Types	14
2.5.1	State-based Convergent Replicated Data Type (CvRDT)	15
2.5.2	Operation-based Commutative Replicated Data Type (CmRDT)	15
2.5.3	Example CRDTs	17
2.5.4	Conclusion	19
2.6	A General-purpose JSON CRDT	19
2.6.1	Structure of a JSON Document	20
2.6.2	Implementation	21
2.6.3	Conclusion	23
2.7	Distributed Programming Languages	23
2.7.1	Argus	24

2.7.2	Distributed Smalltalk	25
2.7.3	Emerald	25
2.7.4	AmbientTalk	26
2.7.5	Geo	27
2.7.6	Lasp	28
2.8	Conclusion	28
3	Strong Eventually Consistent Replicated Objects	
3.1	Formal Definition	32
3.2	Replication and Consistency	33
3.3	Examples	36
3.3.1	Linked List	36
3.3.2	Replicated Linked List	37
3.3.3	Ordered Linked List	40
3.4	Implementation	42
3.4.1	Applying Queries	42
3.4.2	Applying Updates	43
3.4.3	The Commit Operation	47
3.4.4	Time Complexity Analysis	49
3.5	Insights	50
3.6	Conclusion	51
4	CScript	
4.1	Introduction	53
4.2	Motivating Example	54
4.2.1	Functional Requirements	55
4.2.2	Non-functional Requirements	55
4.2.3	Structure of the Application	56
4.3	Language Overview	57
4.4	Services	58
4.4.1	Defining Services	58
4.4.2	Exporting and Discovering Services	60
4.5	Replicas	61
4.5.1	Defining Replicas	61
4.5.2	Consistent Replicas	62
4.5.3	Available Replicas	63
4.6	Limitations	69
5	Implementation	
5.1	The CScript Network and Communication	71
5.1.1	Peer Discovery	72

5.1.2	Network Architecture	72
5.1.3	Publish/Subscribe Mechanism	73
5.1.4	Causal Order Broadcasting	76
5.2	Services And Replicas	77
5.2.1	Parameter Passing Semantics	77
5.2.2	Nesting Replicas	78
6	Evaluation	
6.1	Use Case: Real-time Collaborative Text Editor	83
6.1.1	CScript Implementation	84
6.1.2	JSON CRDT Implementation	90
6.2	Qualitative Analysis	91
6.3	Quantitative Analysis	93
6.3.1	Experimental Set-up	93
6.3.2	Methodology	94
6.3.3	Memory Benchmarks	94
6.3.4	Execution Time Benchmarks	96
6.3.5	Throughput Benchmarks	105
6.4	Conclusion	108
7	Conclusion	
7.1	Our Approach	112
7.1.1	A Distributed Object-oriented Model for Replication	112
7.1.2	Evaluation	114
7.2	Contributions	116
7.3	Limitations And Future Work	116
	References	119

List of Figures

2.1	Illustration of the CAP theorem	10
2.2	Achieving consistency without causal-order broadcasting. The time axis is drawn horizontally, with time increasing from left to right.	16
2.3	Making list insertions and deletions commutative using tombstones. Tombstones are marked in red.	22
3.1	Committing a replica in the face of a network partition.	34
3.2	UML class diagram of the linked list.	36
3.3	Concurrent list insertions at the same position.	41
4.1	Overview of the grocery list application.	54
4.2	UML class diagram of the grocery list application.	56
4.3	UML class diagram of the CScript language.	57
4.4	Concurrently incrementing and deleting grocery items.	65
4.5	Sequence diagram illustrating updates of the grocery application.	68
5.1	A full mesh peer-to-peer CScript overlay network.	72
5.2	Illustration of transitivity in the publish-subscribe mechanism. Publishers and subscribers are indicated with "P" and "S" respectively, grey nodes are neither of both. Disconnected peers are indicated with a dashed red circle and gradient green color. The left node publishes a service, which eventually arrives at the right node, even though both are not online at the same time.	74
5.3	Causal order broadcasting in CScript. The time axis is drawn horizontally, with time increasing from left to right.	76
5.4	Example of a register replica that is nested within a counter replica. The replicas are mutated independently which leads to inconsistencies. Blue arrows indicate network communication.	80
5.5	Avoiding consistency problems with nested replicas by allowing interactions on the containing replica only. Blue arrows indicate network communication.	81

6.1	Class diagram of a text document.	84
6.2	A text document and its tree representation. Numbers indicate the characters' indexes.	87
6.3	A text document and its tree representation. Red numbers indicate index changes compared to Figure 6.2.	87
6.4	A text document and its tree representation. Red number is the position of the newly added character.	88
6.5	Comparison between the memory usage of the SECRO and JSON CRDT text editors. Error bars represent the 95% confidence interval for the average taken from 30 samples. This experiment was performed on a single worker node of the cluster.	95
6.6	Comparison between the list and tree implementations of the SECRO text editor. Error bars represent the 95% confidence interval for the average taken from 30 samples. This experiment was performed on a single worker node of the cluster.	96
6.7	Execution time of a constant time operation in function of the number of experienced operations, for different commit intervals. Error bands represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded.	97
6.8	Time to append a character to the text document in function of the document's length. Error bands represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded. This experiment was performed on the list implementation of the SECRO text editor.	99
6.9	Time to append a character to a document, for the naive SECRO and JSON CRDT text editors. The SECRO version committed the replica after each append operation. Error bands represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded.	100
6.10	Time to append a character to a document, for the list and tree implementations of the SECRO text editor. Replicas were never committed. Error bars represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded.	101

-
- 6.11 Detailed execution time for appending characters to the SECRO text editor. The replica was never committed. The plotted execution time is the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded. 102
- 6.12 Execution time of an operation which appends 100 characters to a text document, in function of the document's length. Replicas were never committed. Error bars represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded. 103
- 6.13 Time to copy a document in function of the number of insertions per operation. The replica was committed after each operation. For each configuration (1, 10, or more insertions per operation) we executed the operation 50 times and measured the copy time of the 50th execution. Measurements are indicated using asterisks. Error bands represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded. 104
- 6.14 A simple linear regression model of the copy time. Error bands represent the 95% confidence interval for the regression line. Measurements are the average of at least 30 samples and are indicated by dots. Samples affected by garbage collection were discarded. 105
- 6.15 Throughput of the naive SECRO and JSON CRDT text editors, in function of the number of concurrent operations. The SECRO version committed the document replica at a commit interval of 100. Error bars represent the 95% confidence interval for the average taken from 30 samples. 106
- 6.16 Throughput of the list and tree SECRO text editors, in function of the number of concurrent operations. Replicas were committed every 50 insertions. Error bars represent the 95% confidence interval for the average taken from 30 samples. . . . 107

Listings

2.1	Using JSON CRDTs to build a grocery list application. <code>doc</code> is a globally available JSON CRDT. Syntax is JavaScript's class syntax in combination with the API proposed by (Kleppmann & Beresford, 2017).	21
2.2	Applying the functional <code>map</code> operation on a set CRDT. Example from (Meiklejohn & Van Roy, 2015).	28
3.1	Pseudocode implementation of the <code>getAt</code> query operation.	38
3.2	Pseudocode implementation of the <code>insert</code> operation.	38
3.3	Precondition for insertions.	39
3.4	Postcondition for insertions.	39
3.5	Pseudocode implementation of the <code>delete</code> operation.	39
3.6	Postcondition for deletions.	40
3.7	Postcondition for ordered concurrent insertions.	41
3.8	Applying a query operation on a replica.	43
3.9	Issuing an update operation.	43
3.10	Receiving updates.	43
3.11	Applying an update operation.	44
3.12	Validating an operation history.	46
3.13	Issuing a commit operation.	47
3.14	Receiving a commit operation.	48
3.15	Committing a replica.	48
3.16	Ignoring operations that apply to elder versions.	49
4.1	Implementation of the grocery service.	58
4.2	Buying a certain quantity of a grocery item.	59
4.3	Exporting grocery services on the network.	60
4.4	Subscribing to grocery services.	60
4.5	Definition of the <code>GroceryService</code> 's replicas.	61
4.6	Implementation of the grocery inventory.	62
4.7	Structure of the available <code>GroceryList</code> data type.	64
4.8	Adding items to a grocery list.	65
4.9	Implementation of the grocery list's <code>bought</code> and <code>delete</code> methods.	66

4.10	Implementation of the grocery list’s get method and the serializable interface.	67
4.11	Listening to update events.	68
6.1	Structure of the naive SECRO text editor.	84
6.2	Implementation of the <code>insertAfter</code> mutator of the naive SECRO text editor.	85
6.3	Implementation of the <code>delete</code> mutator of the naive SECRO text editor.	86
6.4	Structure of the efficient text editor, which organizes its document as a balanced tree of characters.	88
6.5	Inserting a character in a tree-based text document.	89
6.6	Deleting a character from a tree-based text document.	90
6.7	Implementation of a naive text editor using JSON CRDTs.	91
6.8	Character insertions and deletions in the list-based SECRO text editor.	92
6.9	Character insertions and deletions in the list-based JSON CRDT text editor.	92

List of Tables

- 2.1 Classification of distributed programming languages, based on support for availability and consistency. 24
- 3.1 Overview of the adopted pseudocode syntax. 38
- 4.1 Overview of CScript's built-in available data types. 63
- 6.1 Specifications of the Isabelle cluster. 94

1

Introduction

At its inception, the web was mostly comprised of thin clients who displayed the html content provided by servers. A stable connection between clients and servers was vital to the functioning of the application. When losing connectivity the application would abruptly stop working.

Over the past decades hardware evolved at a fast pace, leading to a realm of Internet-connected computing devices. Think for instance of tablets and smartphones. On the other hand, similar evolutions have not been witnessed at a software level. As clients become increasingly mobile, applications can no longer assume connections to be stable. However, applications should remain responsible even in the face of disconnections. Today's expectations are that applications can work offline. This requires offloading computations from the server to the client, leading to what is known as a *thick client architecture*.

Hence, many applications evolved from a thin to a thick client architecture. These two architectures provide different guarantees with regard to the consistency of data. In a thin client architecture clients always fetch the latest information from the server. However, a connection is required in order for the application to work. We say that the application guarantees consistency but not availability. On the other hand, a thick client architecture does not require a constant connection between the clients and the server. However, clients may see outdated information while being offline. This means that the application guarantees availability but not consistency.

To add further complexity, the CAP theorem (Brewer, 2000) states that distributed systems are affected by a trade-off between availability and consistency. This means that programmers need to choose between availability and consistency at the implementation level. This choice is reflected by modern applications which typically maximize offline availability but keep critical parts consistent. A banking application may for instance provide offline functionality but needs to keep the user's balance consistent.

Although most application developers face the aforementioned trade-off, distributed programming languages do not aid the programmer with the development of available and consistent systems.

Our vision is that the next generation of general-purpose distributed programming languages will provide constructs for implementing both available applications and consistent applications. We believe that with the appropriate language constructs and abstractions, the complexities that arise from the trade-off can be hidden behind the language. This frees the programmer from low-level concerns such as replication and data consistency.

To showcase the feasibility of our vision, we design and implement CScript, a distributed programming language with high-level constructs for building both available systems and consistent systems. Using these constructs programmers have a means to develop highly available applications that keep critical parts consistent.

1.1 Research Context

The research conducted throughout this dissertation lies in the intersection of concurrent programming, distributed systems and databases. We briefly describe each field and how it relates to our work.

Concurrent programming Many systems face situations where different computations compete for shared resources. We say that these computations are concurrent. The main challenges faced by concurrent systems are: data consistency, race conditions, deadlocks and livelocks. Our research is concerned with the problem of data consistency. More precisely, components of a distributed system may issue updates concurrently, in which case CScript strives to maintain consistency of the data, to the extent possible.

Distributed systems A distributed system consists of various computers which appear as a single coherent system to the users (Tanenbaum & Van Steen, 2007). Distributed systems are subject to a number of problems, including: different memory access models, latency, partial

failures and concurrency. Our research is mainly related to the problem of partial failures. In essence, CScript strives to keep the system available under partial failures by means of available data structures.

Databases For a couple of decades all research on data consistency focused on implementing distributed transactions within databases. Nowadays, data consistency has become important at a programming language level, since programmers explicitly need to choose between availability and consistency. CScript employs various consistency models to provide well-defined consistency guarantees.

1.2 Problem Statement

The CAP theorem states that distributed systems cannot be available, consistent and partition tolerant at the same time. Instead, distributed systems can achieve only two of these three properties. This forces programmers to choose between availability and consistency, since partition tolerance is a requirement on any real-world distributed system. The problem is that implementing either one, availability or consistency, is intrinsically difficult and there is no language that aids the programmer with this trade-off.

Providing language support for availability and consistency raises a number of essential research questions. First, which type of language constructs are needed to ease the development of both available systems and consistent systems. Ideally, we want to solve the problem at a high-level of abstraction, for instance through available and consistent data structures. However, some problems are bound to low-level solutions, think for instance of the two-phase commit protocol (Bernstein, Hadzilacos, & Goodman, 1987) for implementing distributed transactions. We thus need to shape the required language constructs and additionally define the semantics of these constructs.

Over the past decades, consistency has been extensively studied leading to a wide range of solutions, including consensus algorithms (Lamport, 1998; Ongaro & Ousterhout, 2014) and dedicated language constructs such as far references (Van Cutsem, Mostinckx, Gonzalez Boix, Dedecker, & De Meuter, 2007). However, research is needed to integrate these solutions together with availability in one language. Furthermore, research on availability is still in progress and dedicated language constructs need yet to be developed. Although available data structures exist - e.g. CRDTs (Shapiro et al., 2011b) - they are limited to specific data types and subject to severe restrictions. Hence, developers often resort to ad hoc solutions. This means that developers need to manually implement availability at the core of their applications.

This is a difficult task which requires advanced knowledge of replication and consistency. This leads to our second research question, namely whether a generic available data type that omits the need for ad hoc solutions can be designed.

To summarize, this dissertation is centered around the following research questions:

- RQ. 1** Which language constructs are needed to simplify the development of both available systems and consistent systems, and how can we integrate these constructs in one distributed programming language?
- RQ. 2** Is it possible to design a general-purpose data type that guarantees availability?

1.3 CScript: An Object-oriented Approach

To support programmers with the development of available and consistent distributed systems, we build a novel programming language containing native support for availability and consistency. First, we explore existing language constructs and data types for availability and consistency. We then use these constructs and data types to develop CScript, an extension of JavaScript with first-class replicas and services. Replicas are high-level constructs for developing available and consistent data structures. These data structures can be composed into services, which can be distributed over the network.

Replicas Replicas are a special type of object. Programmers can define two types of replicas: *available* and *consistent* replicas. Available replicas provide availability but guarantee only eventual consistency (Tanenbaum & Van Steen, 2007; Vogels, 2009). This means that users may read different values but eventually at some point in time all users will see the same value. Consistent replicas guarantee strong consistency at the cost of availability. This means that users always have a consistent view on the replica, however, they need to be online to issue updates.

Services CScript allows replicas to be bundled into larger components, called services. Services expose specific functionality and are CScript's unit of distribution. This means that applications can share functionality by exchanging services. CScript ensures that the replicas which make up the service fulfill their availability or consistency guarantee.

In the context of RQ. 2, we notice that the behaviour that is expected from available data types depends on the application at hand. How to handle concurrent operations thus depends on the semantics of the application. For this reason programmers often resort to ad hoc solutions. To address this issue, CScript allows programmers to specify the concurrent behaviour of available replicas through a set of invariants. Afterwards, programmers can implement arbitrary available data structures that behave accordingly to the declared invariants.

1.4 Contributions

The main goal of this dissertation is to design and implement a distributed programming language with built-in support for availability and consistency. We now outline the main contributions of this thesis:

CScript Within the field of distributed programming, we propose CScript, a distributed programming language including available and consistent data types at its core. The novelty of the language is to provide high-level constructs for implementing both available and consistent distributed objects, thereby freeing the programmer from low-level concerns such as replication and data consistency.

SECROs In the context of available data types, we propose SECROs, a general-purpose data type for implementing available data structures. SECROs are used in conjunction with state validators, which are language constructs to specify the concurrent behaviour of SECROs.

State validators come in two forms, preconditions and postconditions, and are associated to the operations of a SECRO. In essence, programmers express invariants over the state of the object. These invariants must hold prior or after the execution of the associated operation.

When facing concurrent operations, SECROs use the state validators to find an execution of the operations that satisfies the declared invariants.

1.5 Thesis Structure

This thesis is structured as follows: first, we describe the findings of our literature study in Chapter 2. Afterwards, Chapter 3 presents our novel SECRO data type, including a formal definition, its implementation and a couple of examples. We then transition to CScript in Chapter 4, describing the constructs for distributed programming provided by the language, by means of

a motivating grocery list example. In Chapter 5 we detail the main design considerations behind the implementation of CScript. Chapter 6 evaluates CScript by means of a comparison between the SECRO data type in CScript and a state-of-the-art approach. First, we compare both approaches from a programming language perspective. Afterwards, we benchmark various performance aspects, including memory usage, execution time and throughput. Finally, Chapter 7 completes this dissertation with a final conclusion, which describes the research performed in the context of this thesis and provides directions for future work.

2

Literature Study

Programmers of distributed systems face a number of problems which are inherent to distribution. Central to this dissertation are the problems of concurrency, partial failures and the lack of a global clock. In order to acquire a broad understanding of the complexities associated with the design and implementation of distributed programming languages, we conduct a literature study reviewing state-of-the-art techniques in distributed programming.

We start with a brief introduction on distributed systems, in Section 2.1, covering the basics of distributed programming and the major challenges that arise from distribution. Following the introduction, Section 2.2 presents the CAP theorem, a conjecture affecting every distributed system. Section 2.3 provides an overview of the different consistency models that are known in the literature. We then turn our attention to the consistency protocols that implement these models, in Section 2.4. In Sections 2.5 and 2.6, we analyze data types that are designed from the ground up to provide specific consistency guarantees. We then present a brief classification of distributed programming languages, in Section 2.7. We discuss each language with regard to its constructs for distributed programming, and how those constructs relate to the CAP theorem. Finally, Section 2.8 concludes this chapter with a brief recapitulation of the insights gathered from this literature study.

2.1 Distributed Systems

The literature provides various definitions of distributed systems. For the purpose of this dissertation, we stick to the definition by (Tanenbaum & Van Steen, 2007): “A distributed system is a collection of independent computers that appears to its users as a single coherent system.”

This definition consists of two important aspects. The first aspect is that a distributed system consists of autonomous components. Components do not share memory and may perform operations simultaneously. Hence, concurrency is inherent to every distributed system. The second important aspect is that a distributed system appears as a single system to its users. This requires the system to mask failures when possible.

Due to the nature of distributed systems, a number of fundamental challenges arise (Coulouris, Dollimore, Kindberg, & Blair, 2012). First, the programmer is forced to deal with different memory access models. Secondly, the programmer must take into account phenomena such as latency, concurrency and partial failures (Waldo, Kendall, Wollrath, & Wyant, 1994). Finally, distributed systems lack a global clock for ordering the events that occur. We outline each of these problems below and analyze their impact on the programmer.

Memory Access Models A distributed system consists of various address spaces, spread over different machines with no form of shared memory whatsoever. Hence, pointers are not shared because they are meaningless in a different address space.

The lack of globally shared memory is an additional source of complexity for the programmer. In the object-oriented programming paradigm, one can imagine the need for sharing an object between different machines. However, sharing requires deep copying the object to the remote address space. This results in two separate instances which need to be manually kept consistent.

Latency Since components of a distributed system do not share memory, they communicate by means of message passing on the network. Those messages must gap the physical distance between the components, which induces a delay on communication. The time it takes to cover that distance is non negligible and commonly referred to as latency.

Because latency is non negligible, programmers must carefully design applications to minimize network communication. Furthermore, delays on communication can be better hidden with an asynchronous communication model, as it does not block the program execution indefinitely.

Concurrency Distributed systems consist of autonomous components that can issue requests concurrently. Therefore, components are a source of concurrency. It is the programmer's task to build a system that remains consistent in the face of concurrent operations, while still achieving a decent amount of parallelism.

Partial Failures In a distributed system individual components, links, and other elements of the network are subject to failures. Since components are autonomous, the failure of one component does not imply the failure of other components. Therefore, failures are said to be partial.

Since partial failures frequently occur in distributed systems, it is the programmer's duty to design and implement a system that is able to cope with these failures, up to a certain degree.

No Global Clock In essence, distributed systems lack a global clock for ordering the events that occur. Additionally, one cannot rely on the physical clocks of individual machines as they are subject to the phenomena of clock drift and clock skew (Tanenbaum & Van Steen, 2007). Therefore, distributed systems typically rely on logical clocks. Logical clocks prescribe a *partial* ordering of the events, that is in accordance with the happened-before relation (Lamport, 1978).

In an attempt to ease the development of distributed applications, early research in distributed object-oriented computing proposed a unified vision of objects (Waldo et al., 1994). The rationale behind this approach is to make abstraction from the location of objects. To this end, the locality of an object is hidden behind the message passing operations. This means that programmers manipulate local and remote objects in the same way.

As described by (Waldo et al., 1994), merely unifying the object models does not suffice, as it does not address the fundamental problems of distributed systems. Instead, distributed programming languages must provide native support to cope with latency, concurrency and partial failures.

2.2 CAP Theorem

From previous section we know that real-world distributed systems are subject to partial failures. Therefore, distributed systems are designed to be robust against partial failures. In practice, coping with partial failures gives rise to an important trade-off, which is formulated by the CAP theorem.

We first explain the different properties (C, A and P) of this theorem. Assume a model in which distributed components can read from and write

to a conceptually shared piece of data. Strong consistency (C) implies that every read observes the latest write. Availability (A) means that components can always issue read and write requests. Hence, the data is available all the time. Finally, partition tolerance (P) means that network partitions do not affect the usability of the system. In other words, components can continue to issue read and write requests when the network is partitioned.

The CAP theorem (Brewer, 2000) states that a distributed system cannot be strongly consistent, available and partition tolerant at the same time. Instead, only two of these three properties can be achieved in combination. Since distributed systems are subject to partial failures, they need to pick partition tolerance. This means that in the face of a network partition the system must choose between availability (AP) or consistency (CP).

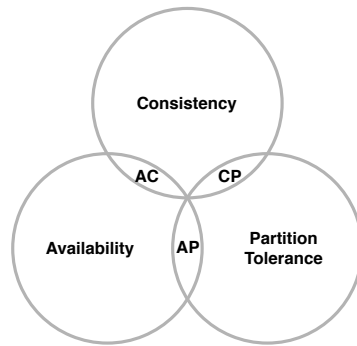


Figure 2.1: Illustration of the CAP theorem

Distributed systems are categorized as AC, AP or CP (cf. Figure 2.1) based on their properties. We briefly discuss each category below.

AC Systems AC systems do not model partition tolerance. In the absence of partitions, the system can guarantee both, availability and consistency. Hence, components can read and write at any moment in time. Additionally, reads always return the most recent information. However, AC systems cannot cope with partial failures. This is undesirable since real-world distributed systems are subject to partitions.

AP Systems AP systems favor availability over consistency. This means that components can read from or write to the data at any time. However, partitions may cause reads to return stale information.

An example of an AP system is the Domain Name System (DNS). The DNS name space consists of various domains and each domain is managed by an authority (Tanenbaum & Van Steen, 2007). When an

authority updates its domain it may take a while before users observe the update. In the meantime, users read stale information.

CP Systems CP systems favor strong consistency over availability. This means that reads always return up-to-date information. Think for instance of distributed transactions, where an update either takes place on all copies or does not take place. Hence, every node of the network has a consistent view of the data. However, updates require agreement (i.e. consensus) between the nodes of the network. Reaching consensus is costly and may be hampered by partial failures, thereby affecting the write availability of the system.

Despite its drawbacks, strong consistency can be a hard requirement, for instance in banking systems. In that case, availability is deliberately discarded in favor of consistency.

Building further on the insights gathered by (Waldo et al., 1994), we argue that distributed programming languages should provide native support for availability (AP) and consistency (CP).

2.3 Consistency Models

Nowadays, distributed systems replicate data to improve the availability, performance and scalability of the system. In the presence of a partial failure it may be possible to fetch the data from another copy, which improves the system's availability. Secondly, distributed systems can reduce access times by placing copies geographically closer to the user, or improve the system's scalability by balancing the work load over replicated nodes.

On the other hand, replication raises potential consistency problems. Users expect read operations to return the most recent information, i.e. the value of the latest write. However, when a copy (aka replica) is updated it becomes different from the other copies. Users may thus observe different values depending on the copy that is read from.

Without a global clock (see Section 2.1) the system cannot determine precisely which write operation is the latest. To address this problem the literature defines a number of consistency models. These models restrict the values that a read operation can return (Tanenbaum & Van Steen, 2007). In essence, a consistency model is a set of rules that provides specific consistency guarantees, if those rules are respected by the programmer.

In what follows we analyze a number of consistency models. First, Section 2.3.1 describes the strong consistency model. Afterwards, Sections 2.3.2

and 2.3.3 present two weaker consistency models: eventual consistency and strong eventual consistency.

2.3.1 Strong Consistency

When dealing with replicated data, the strong consistency model guarantees all copies to be consistent at all times. This implies that after an update completes, all subsequent accesses - possibly by different users - return the updated value. Hence, all users have a consistent view on the system.

Strong consistency requires updates to take effect on all copies before subsequent operations can take place. This means that replicas need to be synchronized. However, synchronization may be hampered by partial failures. In that case, the system does not proceed with the update such that the replicas remain consistent. Hence, when facing a network partition, strong consistency comes at the cost of availability (see CAP theorem in Section 2.2).

2.3.2 Eventual Consistency

We previously explained that distributed systems replicate data to improve the performance and scalability of the system. However, keeping all copies strongly consistent is expensive (see Section 2.3.1). Furthermore, we know from the CAP theorem (Section 2.2) that a distributed system cannot guarantee both availability and strong consistency.

To meet these problems one can relax the consistency guarantees that are expected from the system. One such model is *eventual consistency* (EC). Eventual consistency (Tanenbaum & Van Steen, 2007; Vogels, 2009) prescribes that updates eventually propagate to all copies. In the meantime, users may observe temporal inconsistencies.

Notice, however, that concurrent writes cause additional problems since the copies need to agree which write operation is the latest. This requires synchronization of the copies.

2.3.3 Strong Eventual Consistency

(Shapiro et al., 2011b) proposed strong eventual consistency (SEC), a special form of eventual consistency which guarantees replicas that received the same updates, possibly in a different order, to be in a consistent state. This property is called *strong convergence*.

Strong convergence implies that replicas converge without synchronization. Hence, it suffices to propagate the updates between the replicas. As

such, all replicas experience the same updates and thus converge towards the same state.

The fact that replicas converge without synchronization is a major advantage of SEC over EC. First, it eases the development of eventually consistent systems. Second, systems achieve better performance and scalability.

2.4 Consistency Protocols

The previous section described the consistency guarantees provided by different consistency models. We now turn our attention to consistency protocols, which describe the implementation of these models. We follow the same structure as the previous section and set of with the implementation of strong consistency, followed by eventual and strong eventual consistency.

2.4.1 Strong Consistency

As previously explained, the strong consistency model guarantees all users to have a consistent view on the data. Hence, updates require agreement of the nodes that make up the system, such that all copies are updated consistently. Since copies are not allowed to diverge, we are essentially applying a *pessimistic replication* strategy in combination with a distributed consensus algorithm.

In theory, there is no algorithm that can always reach consensus in a distributed system (Fischer, Lynch, & Paterson, 1985). This results from the fact that distributed consensus algorithms cannot guarantee termination. However, in practice some consensus algorithms have proven to work well, including Paxos (Lamport, 1998) and the two-phase commit protocol (Bernstein et al., 1987).

2.4.2 Eventual Consistency

As explained in Section 2.3.2, eventual consistency is a relaxation of the consistency guarantees, which allows replicas to exhibit temporal inconsistencies. However, if we stop updating the data, eventually all replicas reach a consistent state again.

In practice, eventual consistency boils down to using *optimistic replication*. Optimistic replication is a replication strategy that allows replicas to temporarily diverge. Replicas can thus be updated immediately without requiring prior consensus of the system. However, concurrent updates may

lead to inconsistent replicas. Eventually consistent systems solve this problem by means of a conflict resolution strategy which synchronizes the replicas. Many of these strategies discard updates in order to solve the conflict. For instance, if two updates conflict, discarding one update is all it takes to solve the conflict. Although this approach seems odd it is widely used because of its simplicity, for example in “last-writer-wins” strategies (Burckhardt, 2014).

2.4.3 Strong Eventual Consistency

Remember that strong eventual consistency (SEC) is a special form of eventual consistency, which guarantees replicas to be consistent if they received the same updates (strong convergence). An important subtlety is that the order in which updates are delivered is immaterial and may be different at all replicas. Intuitively, the strong convergence property holds for commutative operations.

Designing data types for commutativity is the fundamental idea behind *conflict-free replicated data types* (CRDTs). CRDTs are abstract data types that implement strong eventual consistency.

A main advantage of SEC is that it suffices to propagate the updates to all replicas. Eventually, all updates are delivered at all replicas which guarantees the replicas to be in a consistent state. Hence, this approach does away with synchronization, yielding higher performance and scalability. For this reason, SEC is of special interest to this dissertation. We dedicate the following section to CRDTs.

2.5 Conflict-free Replicated Data Types

Conflict-free replicated data types (CRDTs), proposed by (Shapiro et al., 2011b), are abstract data types that implement strong eventual consistency. CRDTs leverage some mathematical properties to ensure conflict freeness. In the absence of conflicts, replicas that experienced the same updates - possibly in a different order - are in a consistent state. Hence, programmers do not need to rely on conflict resolution strategies to ensure eventual consistency. This makes CRDTs interesting for the development of large-scale applications.

Conflict-free replicated data types come in two variants: state-based and operation-based CRDTs. The former typically allows for simple reasoning, whereas the latter is often preferred in practical systems. Both are equivalent which is particularly useful. We refer the interested reader to the proof of equivalence in section 3.2 of (Shapiro et al., 2011b).

2.5.1 State-based Convergent Replicated Data Type (CvRDT)

A state-based CRDT is a tuple (S, s_i, Q, U, m) , consisting of the replica's state domain¹ S , the replica's current state s_i , a set of (side-effect free!) query methods Q , a set of update methods (aka mutators) U and a merge method $m : S \times S \rightarrow S$. A replica's public interface typically consists of Q and U and should not be circumvented. Programmers use the query methods Q to read (parts of) the internal state. On the other hand, update methods U mutate the replica's internal state.

When experiencing a mutation, the replica transitions from its current state s_i to the new state s_{i+1} , and disseminates the resulting state s_{i+1} over the network. To process incoming state updates, replicas merge the received state s_r with their own state s_i , resulting in a new state $m(s_i, s_r) = s_{i+1}$.

To achieve strong eventual consistency, CvRDTs rely on the mathematical property of a join semilattice, that is a partially ordered set equipped with a least upper bound (LUB) for all value pairs (Davey & Priestley, 2002). First, the CvRDT's state domain S must form a join semilattice, with a partial order denoted \leq . Second, all update methods U must result in a monotonically non-decreasing state. The resulting state subsumes the original state, $s_i \leq s_{i+1} = u(s_i)$. Finally, the merge method m computes the LUB of its input states and must be associative, commutative and idempotent. As a result of the above properties, replicas that received the same updates converge towards the LUB of the involved states.

Although state-based CRDTs guarantee SEC, they suffer from a major drawback. Every mutation requires the entire resulting state to be exchanged over the network. As a solution, (Almeida, Shoker, & Baquero, 2015) propose δ -CRDTs, an adaptation of CvRDTs where delta-states are exchanged, resulting in smaller messages.

2.5.2 Operation-based Commutative Replicated Data Type (CmRDT)

An op-based CRDT is a tuple (S, s_i, Q, U) that does not define a merge method. Again, S is the object's state domain, s_i the current state, Q a set of accessors and U a set of update methods. When an update causes the object to transition to a new state, the operation is broadcasted to all replicas. Since query operations do not incur side effects, only update operations are exchanged.

¹The set of all possible states.

We emphasize the fact that (Shapiro et al., 2011b) assume the operations to be delivered in a causal order. Still, concurrent operations exhibit no ordering and arrive in an arbitrary order that may be different at all replicas. Therefore, op-based CRDTs guarantee SEC by imposing commutativity of these operations. As such, the order in which replicas apply the operations does not affect the final outcome. For this reason, op-based CRDTs are often called “Commutative Replicated Data Type” (CmRDT).

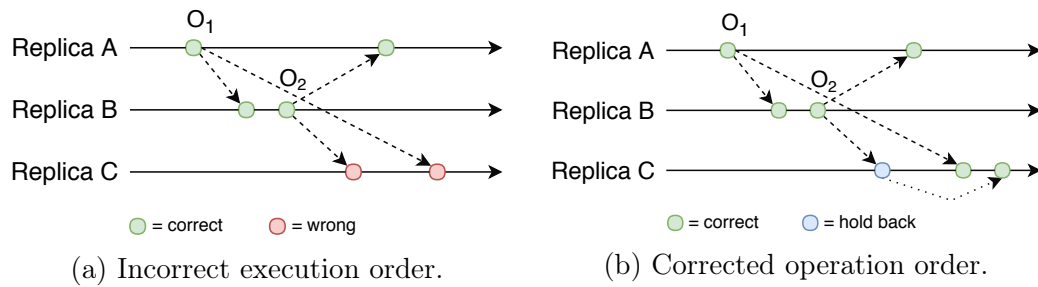


Figure 2.2: Achieving consistency without causal-order broadcasting. The time axis is drawn horizontally, with time increasing from left to right.

Often times the communication mechanism does not guarantee causal-order broadcasting. This is problematic since operations do not necessarily arrive in the order they occurred. Hence, the replicas will converge only if *all* operations are commutative. In practice, it is often not possible to design all operations to commute. However, with causal order broadcasting, only the operations which may occur concurrently need to be commutative. To illustrate the aforementioned problem, Figure 2.2a depicts the case where operation O_2 arrives at replica C, before its dependency O_1 . This forms a problem if O_1 and O_2 do not commute. Such anomalies occur due to phenomena such as congestion and network partitions.

To guarantee SEC in the absence of a causal-order broadcasting mechanism, op-based CRDTs are extended with a dependency set maintaining the object’s causal dependencies. The execution of an update method generates a unique identifier that is added to the object’s dependency set. When broadcasting an operation, its unique ID as well as the dependency set are included in the message. Upon receiving an operation, the operation is held back until all dependencies are met (cf. Figure 2.2b). As such, we simulate causal-order broadcasting by manually tracking causal dependencies. However, a replica’s dependency set grows monotonically over time, eventually becoming a problem as messages get longer.

2.5.3 Example CRDTs

(Shapiro, Preguiça, Baquero, & Zawirski, 2011a) introduce a portfolio of CRDTs, including counters, registers, sets, graphs and other data types. To illustrate some frequently recurring design strategies for CRDTs, we now review a few of these data types. A basic understanding of these design strategies is needed to understand the remainder of this dissertation. Notice that we describe a state-based implementation of the examples because it provides a better understanding of the data type compared to its op-based implementation.

Grow-only Counter The state domain of a counter consists of all positive natural numbers, \mathbb{N}^+ . Furthermore, (\mathbb{N}^+, \leq) forms a join-semilattice where the join of two natural numbers returns the bigger one, $i \vee j = \max\{i, j\}$ where $i, j \in \mathbb{N}^+$.

Similarly, a grow-only counter (G-Counter) is a tuple:

$$(S, \vec{V}, \{value\}, \{increment\}, merge)$$

The counter's internal state consists of a vector $\vec{V} = (c_1, c_2, \dots, c_n)$, maintaining a counter value for each of the n nodes in the cluster. Therefore, the counter's state domain S is the set of all counter vectors: $\{\vec{V} \mid \|\vec{V}\| = n \wedge \forall i \in [1, n] : \vec{V}[i] \in \mathbb{N}^+\}$. The update method `increment` increments the node's own vector entry, for node i this yields: $\vec{V}[i] := \vec{V}[i] + 1$. The query method `value` returns the actual counter value, which is the sum of the vector entries, $\sum_{i=1}^n \vec{V}[i]$. Finally, merging two G-Counters computes the LUB of the corresponding vector entries, $merge(\vec{V}_1, \vec{V}_2) = \forall i \in [1, n] : \vec{V}_1[i] \vee \vec{V}_2[i]$.

Positive-Negative Counter For regular counters additional problems arise since decrement operations violate the monotonicity of the semilattice. In other words, the state resulting from a decrement operation does not subsume the original state, $s_i \not\leq s_{i+1} = u(s_i)$ where $u = decrement$.

A counter CRDT supporting both, increments and decrements, is called a Positive-Negative counter (PN-counter). A PN-counter is a tuple:

$$(S, (P, N), \{value\}, \{increment, decrement\}, merge)$$

The internal state consists of two G-Counters: P and N . The P counter counts increments whereas N counts decrements. Therefore,

the counter's actual value is $P - N = \sum_{i=1}^n P[i] - \sum_{i=1}^n N[i]$. Finally, merging two PN-counters consists of merging the corresponding P and N counters.

This example showcased the use of two grow-only counters to circumvent the decreasing nature of a decrement operation. In practice, many complex CRDTs are a combination of basic CRDTs.

Grow-only Set Other classical examples include the implementation of sets. The simplest form is a grow-only set CRDT (G-Set):

$$(S, s, \{\text{contains}\}, \{\text{add}\}, \text{merge})$$

The state domain S consists of all possible sets, and the subset relation \subseteq defines the lattice's partial order \leq . Initially, the internal state s is the empty set \emptyset . `contains` checks for the presence of an element in the set. The `add` operation adds an element e to the set s , resulting in a new set that subsumes the old, $\text{add}(e) = \{s, e\} \supseteq \{s\}$. Hence, the `add` operation is monotonically non-decreasing. Finally, `merge` computes the LUB of two G-Sets, which is defined to be the union: $\text{merge}(s_1, s_2) = s_1 \cup s_2$.

Two-phase Set Extending a set to support removal of elements is problematic, since deletion infringes the monotonicity of the semilattice. Similarly to PN-counters, this problem can be circumvented by carefully designing the set as a combination of two G-Sets. The resulting data type is a two-phase set (2P-Set) CRDT:

$$(S, (A, R), \{\text{contains}\}, \{\text{add}, \text{remove}\}, \text{merge})$$

Set A is used to add elements, whereas R acts as a remove set. Upon removing an element, `remove` adds the element to R . An element is considered present in the set if it occurs in $A \setminus R$. By design, elements occurring in R are permanently deleted. Finally, the merge procedure computes the LUB of two 2P-Sets. To this end, `merge` computes the union of the corresponding add (A) and remove (R) sets.

This example showcased the use of *tombstones*, namely the elements in R , to delete elements. Due to the monotonicity condition, CRDTs cannot delete elements. Instead, tombstones are used to mark elements as deleted. Tombstones are a typical design strategy for simulating delete functionality in CRDTs.

2.5.4 Conclusion

Remember that eventually consistent systems may face conflicting updates. To deal with conflicts, the system relies on a conflict resolution strategy which synchronizes the replicas. However, synchronizing the replicas comes at a cost. Therefore, SEC goes a step further and avoids the need for synchronization altogether.

A protocol for SEC are conflict-free replicated data types. CRDTs are abstract data types that avoid conflicts by design. Hence, there is no need for conflict resolution strategies. This results in better performance and scalability.

The literature distinguishes between state-based and op-based implementations for CRDTs. The former disseminates the entire state, whereas the latter disseminates the operations. In general, the op-based style reduces network traffic whereas the state-based style avoids repeated computations. Hence, choosing an implementation depends on the size of the state and the cost of operations.

However, CRDTs exhibit some major drawbacks. The literature provides only a limited portfolio of basic conflict-free data structures. This forces developers to resort to ad hoc, application-specific mechanisms for eventual consistency. These mechanisms are error-prone and result in brittle systems (Shapiro et al., 2011b; Almeida et al., 2015; Kleppmann & Beresford, 2017).

Furthermore, complex systems require tailored CRDTs. However, CRDTs are not generally applicable as they require the operations to commute, or the state to form a join-semilattice. This renders the design of custom CRDTs a challenging task which requires advanced knowledge of replication and consistency techniques.

A final problem arises from the use of tombstones to simulate deletions. Because elements are never actually deleted and tombstones are continuously added, memory usage grows unbounded. Solving this problem requires a distributed garbage collection algorithm that is able to recognize and appropriately remove tombstones. However, tombstone removal is left to the programmer.

2.6 A General-purpose JSON CRDT

Up till now, literature has focused on developing new conflict-free replicated data types, including counters, sets, graphs and so on. However, considerable research efforts are needed to solve the aforementioned applicability issue of CRDTs. With this goal in mind, (Kleppmann & Beresford, 2017) proposed a

general-purpose conflict-free replicated JSON data type, which is a generalization of CRDTs. The proposed CRDT resembles generic JSON documents, i.e. a collection of lists and maps that can be arbitrarily nested. As such, the JSON CRDT reduces the need for ad hoc, application-specific strategies towards eventual consistency.

As explained in Section 2.4.2, some conflict resolution strategies discard updates to solve conflicts. (Kleppmann & Beresford, 2017) consider this undesirable as it incurs a loss of data. Therefore, a major design principle of the JSON CRDT is not to drop updates.

2.6.1 Structure of a JSON Document

A JSON CRDT exhibits a structure similar to a JSON document, that is a tree of branch and leaf nodes. Branch nodes consist of lists and maps, whereas leaf nodes are primitive values. We briefly summarize the different components of a JSON document:

Primitives The proposed data structure supports four primitive types: `null`, booleans, numbers and strings. Values are primitives, lists and maps.

List A list is a sequence of ordered values. Elements contained by the list are the node's children. The list interface allows programmers to fetch the element at a given position, to insert a value after a certain element or to delete a specific element.

Map A map is a dictionary of key-value pairs that allows bindings to be added, modified and deleted dynamically. Keys are immutable and the children exhibit no ordering. If the keys represent field names, this data structure is better known as an object.

The novelty of JSON CRDTs is that the aforementioned components can be nested. Hence, developers can build complex structures by composing lists and maps in arbitrary ways. To illustrate this, Listing 2.1 shows the implementation of a custom `GroceryList` data type. To create a grocery list with a given name the constructor initializes a new key-value pair in the global JSON CRDT (Line 3). The list's name is the key and the value is an empty list. To add items the `addItem` method fetches the grocery list (Line 7) and prepends the item to the list (Lines 8 and 9). To delete an item from a grocery list, the `removeItem` method searches the item in the list (Lines 14 to 18) and deletes it (Line 16).

```

1  class GroceryList {
2      constructor(name) {
3          doc.get(name) = [];
4      }
5
6      addItem(list, item) {
7          doc.get(list)
8              .idx(0)
9              .insertAfter(item);
10     }
11
12     removeItem(list, item) {
13         var lst = doc.get(list);
14         for (var i = 1; i <= lst.length; i++) {
15             if (lst.idx(i) == item) {
16                 lst.idx(i).delete();
17             }
18         }
19     }
20 }

```

Listing 2.1: Using JSON CRDTs to build a grocery list application. `doc` is a globally available JSON CRDT. Syntax is JavaScript’s class syntax in combination with the API proposed by (Kleppmann & Beresford, 2017).

The above code snippet showcased the simplicity of the model. We defined a custom data type that guarantees SEC, without having to implement conflict resolution ourselves. Therefore, JSON CRDTs are said to be general-purpose.

2.6.2 Implementation

(Kleppmann & Beresford, 2017) detail an op-based implementation of the JSON CRDT. Hence, concurrent operations are designed to be commutative. We briefly analyze the different aspects of this implementation, paying special attention to the commutativity design.

List Insertions Lists allow elements to be inserted at arbitrary positions.

Due to the distributed nature of the system, two or more elements can be concurrently inserted at the same position in a list. To guarantee a consistent order across all replicas, the elements are ordered based on their unique ID.

Assignments Maps are the only data structure that allow values to be reassigned. Entries of a map are multi-value registers (mv-registers) which hold a value. A value can be updated by reassigning the register.

A mv-register is a special register CRDT, designed to keep all values in the face of concurrent assignments. The mv-registers internally used

by maps differ from traditional mv-registers described in the literature. Upon concurrent assignments, the register attempts to merge the values in a meaningful manner. When some values cannot be merged, the register stores them separately.

Merge Procedures (Kleppmann & Beresford, 2017) define strategies for merging lists and maps meaningfully. Two or more lists are merged by appending them. To break ties consistently, lists are appended according to a total order (e.g. based on IP address). Notice that identical values (lists, maps or primitives) are not merged, since they are already equal.

Similarly to lists, the merger of two or more maps consists of the union of their bindings. If a given key occurs more than once, the values are in turn merged.

Deletions Complex data types (i.e. lists and maps) use tombstones to provide delete functionality. Hence, deleting an element merely marks the element as deleted. This guarantees delete to commute with the other operations. Figure 2.3 depicts the insertion of an element Y after X , while concurrently X is deleted. The order in which the operations are applied on the linked list is immaterial. Since delete does not actually remove the element, Y can always be inserted after X (see Figure 2.3b).

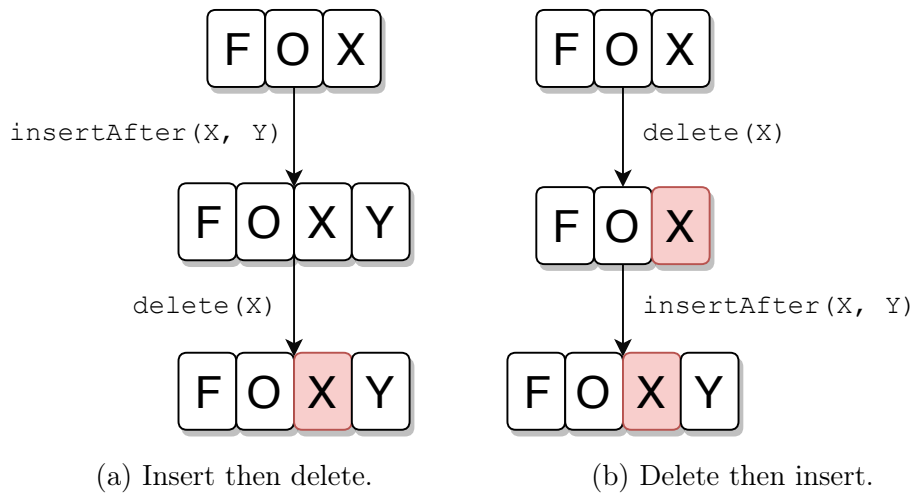


Figure 2.3: Making list insertions and deletions commutative using tombstones. Tombstones are marked in red.

Besides the operations discussed so far, complex data types can also be cleared. Clearing a list or map is done by reassigning the register to

an empty list, respectively an empty map. Empty list and map assignments are treated specially. Instead of overwriting the register - which is not commutative - the individual elements of the list, respectively the map, are deleted. As such, complex data types can be “cleared” in a commutative manner.

Nesting JSON CRDTs cannot be nested, however, the lists and maps that make up the CRDT can be nested (see Section 2.6.1). Therefore, cursors are used to unambiguously identify positions within the document. A cursor describes the path from the root of the document to the node at hand.

When an operation executes at a certain position in the document, a cursor identifying that position is created. The replica then disseminates the operation and its cursor to the other replicas.

2.6.3 Conclusion

This section described JSON CRDTs, a *generic* CRDT which partly solves the applicability problem of CRDTs. Using JSON CRDTs, programmers can create new eventually consistent data types without having to deal with conflict resolution. These data types can be of arbitrary complexity, since lists and maps can be nested.

The problem is that JSON CRDTs exhibit severe shortcomings. Different applications have different semantics, therefore requiring different behaviours. However, programmers cannot define application-specific merge procedures as in Bayou (Terry et al., 1995).

Secondly, in the absence of applicable merge procedures, individual values are stored by the register. This technique, known as “semantic resolution” (Meiklejohn & Van Roy, 2015), leaves conflict resolution to the programmer, which is a difficult task.

2.7 Distributed Programming Languages

Recall that our thesis focuses on programming language abstractions for availability (AP) and consistency (CP). Therefore, we conclude this literature study with a brief classification of distributed programming languages. The languages are classified and analyzed according to the level of support they provide for availability and consistency.

The classification in Table 2.1 contains a number of early influential distributed programming languages, including Argus, Distributed SmallTalk

and Emerald. Additionally, we investigated a number of recent research efforts, including AmbientTalk, Lasp and Geo.

Language	AP	CP
Argus	✗	Actions
Distributed Smalltalk	Copy	RMI
Emerald	Move	RMI
AmbientTalk	Isolates	Far References
Geo	Caching	Linearizability & Cache Coherence Protocol
Lasp	CRDTs	✗

Table 2.1: Classification of distributed programming languages, based on support for availability and consistency.

2.7.1 Argus

Argus, proposed by (Liskov, 1988) was developed to facilitate the implementation of object-oriented distributed systems. The authors realized that concurrency and partial failures pose major difficulties for the implementation of distributed systems. Therefore, Argus includes high-level constructs that free the programmer from consistency problems, race conditions and other typical problems.

In Argus, distributed systems are built around *guardians*, which form the unit of distribution. A guardian is a black box encapsulating one or more objects, called resources. Programmers cannot fiddle with a guardian’s internal state since all accesses and invocations are regulated by the guardian’s API. Methods of a guardian are called handlers, and each handler invocation runs in a separate process.

A second remarkable abstraction are actions. Actions are atomic methods that exhibit two important properties: *serializability* and *totality*. Serializability guarantees the outcome of a concurrent execution to equal a sequential execution. Hence, actions are robust against concurrency. On the other hand, totality guarantees the activity to either complete entirely or not at all, in which case the action is said to be aborted.

In addition to regular objects, Argus provides atomic objects. Whereas regular objects implement methods, atomic objects implement actions. Programmers benefit from the fact that atomic objects synchronize their actions

and recover from aborted actions. Hence, if a partial failure causes an action not to complete entirely, the action is aborted and the atomic object recovers to its previous consistent state. In order to guarantee atomicity a two-phase commit protocol is used (Bernstein et al., 1987).

To conclude, Argus provides the programmer with actions, a special type of method that guarantees strong consistency. Actions are thus a CP language construct, as shown in Table 2.1. On the other hand, Argus does not provide language constructs for availability.

2.7.2 Distributed Smalltalk

Distributed Smalltalk (DS), presented by (Bennett, 1987), is a variant of Smalltalk that aims at improving the communication and interaction between different Smalltalk nodes. DS extends Smalltalk with transparent remote invocations, distributed garbage collection and object mobility.

Programmers can access remote objects and interact with the objects by means of remote method invocations (RMI). The language thus hides the locality of an object, making invocations on remote objects transparent to the user.

RMI can be seen as a consistency mechanism as it guarantees a single, consistent, instance of the object. However, RMI introduces a single point of failure which tremendously impacts the availability of the object.

Remarkably, DS also addresses the mobility of objects. Programmers can use the move and copy primitives to move or copy an object to another node respectively. By incorporating native support for replication through the copy operation, DS thus lays the foundations for available systems.

However, DS does not provide any consistency guarantees on copied objects. Hence, copies of an object can co-exist without being consistent. A second drawback is the limited support for object mobility. Moving objects requires the receiver to know the object's entire class hierarchy. This may lead to subtle compatibility problems, due to inconsistencies between the sender's and the receiver's classes. Regarding compatibility, the language performs some basic checks but cannot provide additional guarantees.

In conclusion, distributed Smalltalk provides native support for both availability and consistency. We explained that RMI is a consistency mechanism whereas the copy primitive is a replication mechanism (see Table 2.1).

2.7.3 Emerald

Emerald is a distributed object-oriented programming language, primarily designed to experiment with the mobility of objects in a distributed environ-

ment (Jul, Levy, Hutchinson, & Black, 1988). To this end, Emerald includes extensive language support for object mobility.

In Emerald, inter-object communication is carried out by remote invocations. Similarly to Distributed Smalltalk, remote invocations constitute a communication mechanism that guarantees consistency in the absence of replication. Therefore, we list remote invocations as a CP construct in Table 2.1. Notice that Emerald’s remote invocations differ from traditional RMI, the details of which are immaterial to the present discussion.

Emerald differentiates itself from Distributed Smalltalk, in that it provides extensive support for object mobility. Programmers can migrate objects from one node to another using the `move` primitive. By migrating objects one can improve failure coverage, therefore, we list `move` as an AP language construct in Table 2.1. Object migration can also increase performance by placing the data geographically closer to the user. In addition, objects can be attached to form groups of objects that move together. Other functionality includes fixing and unfixing the location of objects at particular nodes.

Besides the mobility model, Emerald introduces novel parameter passing semantics for remote method invocations. The fundamental idea is to move argument objects to the callee. As such, all arguments are locally available when the method executes. This avoids a number of costly additional remote invocations that would arise from interactions with the arguments. Programmers can choose to return the objects to the caller after the method’s execution or to keep the objects at the receiver. The former is named “call-by-visit” whereas the latter is named “call-by-move”.

2.7.4 AmbientTalk

AmbientTalk, originally described by (Van Cutsem et al., 2007), is a distributed object-oriented programming language featuring an actor-based, event-driven concurrency model.

AmbientTalk has a built-in publish-subscribe mechanism for service discovery. This mechanism allows programs to discover objects that export specific services in an ad hoc network. Upon discovering a service, the user acquires a remote reference to the discovered service object. This reference works only via asynchronous message passing, and is called a *far reference*. Note that far references are resilient to partial failures. In the face of a network partition, the far reference buffers all messages until the remote object becomes available again, at which point the buffered messages are flushed.

Conceptually, remote objects behave as shared, consistent objects, even though only a single instance exists underneath. Therefore, far references are a consistency mechanism (see Table 2.1). Analogous to the previous

languages, the remote object forms a single point of failure which affects the object’s availability.

AmbientTalk also includes native support for replication through isolates. Isolates are self-contained first-class objects which cannot access their surrounding lexical scope. This allows isolates to be passed by value and as such form the basis for replication. Hence, isolates are a language construct for availability, listed in Table 2.1.

Finally, we observe that AmbientTalk includes native support for both, strong consistency (far references) and availability (isolates). However, distributed objects cannot guarantee both properties (see Section 2.2) and AmbientTalk provides no middle ground. Hence, if availability is key and some form of consistency is expected, the programmer will need to manually implement an eventually consistent layer atop isolates.

2.7.5 Geo

Geo, recently proposed by (Bernstein et al., 2017), is an actor system specifically designed for geographically distributed actors. Geo targets actors in large cluster environments, such as datacenters, which can be separated by a considerable physical distance. Due to the physical distance between datacenters, inter-actor communication may entail high latency. Therefore, Geo uses caching and replication techniques to hide the latency and benefit from data locality where possible.

Geo supports “single-instance” and “multi-instance” caching policies for actors. The former ensures a single consistent instance of the actor, by caching the actor’s state only at one node. The latter replicates the actor to every cluster, by caching the actor’s state in one node of every cluster.

Although caching improves availability and performance, caching can lead to stale information. Therefore, Geo includes native support for various consistency models (see Section 2.3). First of all, actors support *linearizable* reads and writes. Linearizability guarantees strong consistency of single-instance actors. For “multi-instance” caching policies, Geo coordinates the actors to act as a single latest version, using a distributed cache coherence protocol.

In contrast to the previously analyzed languages, Geo provides a “Versioned API” which is a middle ground between availability and consistency. The Versioned API is a variation on the global sequence protocol (Burckhardt, Leijen, Protzenko, & Fähndrich, 2015), that guarantees eventual consistency of the actors.

To summarize, we explained that Geo actors can be replicated to different clusters by means of the multi-instance caching policy. To keep these

replicated actors strongly consistent, Geo provides a distributed cache coherence protocol. Weaker consistency models can also be adopted using the Versioned API. Hence, Geo is a full-fledged actor system with native support for availability as well as strong and eventual consistency. Support for availability and strong consistency is described in Table 2.1.

2.7.6 Lasp

Lasp (Meiklejohn & Van Roy, 2015) is a distributed programming language designed to simplify the development of large-scale distributed systems. To this end, Lasp provides conflict-free replicated data types as first-class values.

Developing complex coordination-free systems requires the composition of CRDTs, which is challenging and error-prone. To cope with this problem, Lasp supports composition of CRDTs through a functional API. The functional API includes the `map`, `filter` and `fold` primitives. Each functional primitive takes a CRDT as input and produces a new CRDT. Hence, functional primitives can be composed in arbitrary ways, allowing complex CRDT transformations through an intuitive API.

```
1 {ok, S1} = lasp:declare(riak_dt_orset),  
2 {ok, _} = lasp:update(S1, {add_all, {1,2,3}}, a),  
3 {ok, S2} = lasp:declare(riak_dt_orset),  
4 {ok, _} = lasp:map(S1, fun(X) -> X * 2 end, S2).
```

Listing 2.2: Applying the functional `map` operation on a set CRDT. Example from (Meiklejohn & Van Roy, 2015).

Listing 2.2 illustrates the functional API by means of an example. First, the example creates an empty set CRDT and populates it with the numbers 1, 2 and 3 (Lines 1 and 2). Afterwards, Line 4 maps over the set multiplying each number by two, and stores the resulting set CRDT in `S2`.

To conclude, Lasp supports intuitive composition of CRDTs in a way that guarantees strong eventual consistency. However, Lasp does not provide native support for the strong consistency model. Therefore, we classify Lasp as an AP language in Table 2.1.

2.8 Conclusion

The development of distributed systems is affected by the CAP theorem (Brewer, 2000) which implies a trade-off between strong consistency and availability. Therefore, many applications favor availability over strong consistency and instead guarantee a weaker form of consistency, known as eventual consistency. Strong eventual consistency (SEC), proposed by (Shapiro

et al., 2011b), is a variation on eventual consistency that avoids the need for synchronization, yielding better scalability and performance. To the best of our knowledge, conflict-free replicated data types (CRDTs) currently remain the only mechanism to guarantee strong eventual consistency.

The potential of SEC raises special interest from large-scale distributed systems. The integration of this consistency model into real-world applications is however hampered by significant barriers. First, building complex systems requires application-specific CRDTs that are tailored to the needs of the application. However, conflict-freeness requires the operations to be commutative. This is a strong requirement which affects the applicability of CRDTs. Hence, CRDTs are not flexible, which renders the design of custom CRDTs particularly challenging.

In an attempt to circumvent the aforementioned problem, programmers may resort to JSON CRDTs (Kleppmann & Beresford, 2017), which are so called general-purpose CRDTs. However, JSON CRDTs exhibit hardcoded concurrent behaviours which contradicts their general applicability. Furthermore, JSON CRDTs resort to semantic resolution (see Section 2.6.3) when falling short of applicable merge procedures. This implies that programmers need to manually solve the conflict. We argue that semantic resolution is undesirable as it places the burden of conflict resolution on the programmer.

Finally, our classification of distributed programming languages revealed that most languages only provide limited support for availability and consistency. With the exception of Geo, none of the presented programming languages includes native support for eventual consistency, let alone strong eventual consistency.

3

Strong Eventually Consistent Replicated Objects

Although distributed systems could greatly benefit from CRDTs, their integration within real-world systems is challenging. In this chapter we explore the integration of strong eventual consistency (SEC) with the object-oriented programming paradigm, by introducing strong eventually consistent replicated objects (SECROs).

SECROs provide a flexible interface which does not require the operations to be commutative, while still guaranteeing SEC. To resolve conflicts SECROs use a semi-automatic conflict resolution strategy that is based on application-level semantic information declared by the programmer.

We start with a formal definition of strong eventually consistent replicated objects in Section 3.1. Following this definition, Section 3.2 focuses on replication and outlines the provided consistency guarantees. We then present two examples that differentiate SECROs from traditional approaches, in Section 3.3. Section 3.4 presents a pseudocode implementation of the SECRO data type, followed by a time complexity analysis. Afterwards, Section 3.5 outlines a number of advanced insights. Finally, Section 3.6 concludes this chapter with a synopsis of the SECRO data type.

3.1 Formal Definition

A strong eventually consistent replicated object (SECRO) is a tuple (v_i, s_0, s_i, Q, M, h) , consisting of a version number v_i , the object's initial state s_0 , the current state s_i , a set of (side-effect free!) query operations Q , a set of mutators M and an operation history h . The object's public interface consists of Q and M . Programmers access (parts of) the internal state through the query operations and update the internal state using mutators. Notice that mutations do not affect the initial state s_0 . Instead, mutations update the current state $s_i \xrightarrow{m} s_{i+1}$ and are added to the operation history.

Building further on this definition, we now outline a number of fundamental concepts.

Mutators A *mutator* is a triple (o, p, a) consisting of an update operation o , a precondition p and a postcondition a . The update operation is a function: $o : A_1 \times \dots \times A_n \rightarrow R$, that takes n arguments (where A_i denotes the type of the i -th argument) and returns a result of type R .

In the face of concurrent operations the SECRO data type relies on preconditions and postconditions to determine a conflict-free ordering of the operations. Preconditions and postconditions are so-called *state validators*.

State Validators Programmers define state validators to specify a data type's behaviour in the face of concurrency. State validators are declarative rules that are associated to certain operations. Those rules express invariants over the state of the object and as such translate the semantics of the operations. State validators come in two forms, namely preconditions and postconditions.

Preconditions Preconditions specify invariants that must hold prior to the execution of their associated operation. As such, preconditions approve or reject the state before applying the actual update. In case of a rejection, the update is aborted.

Mathematically, we define a precondition to be a function $p : S \times A_1 \times \dots \times A_n \rightarrow \mathbb{B}$ where S is the SECRO's state domain. The precondition thus takes $n + 1$ arguments, namely the object's current state s_i followed by the n arguments passed to the update operation o , and returns a boolean indicating whether to approve or reject the state.

Postconditions Postconditions specify invariants that must hold after the execution of their associated operation. In contrast to

preconditions, an operation's associated postcondition does not execute immediately. Instead, the postcondition executes after all concurrent operations complete. As such, postconditions approve or reject the state that results from a group of concurrent, potentially conflicting operations.

Mathematically a postcondition is a function $a : S \times S \times A_1 \times \dots \times A_n \times R \rightarrow \mathbb{B}$ where S is the SECRO's state domain. The postcondition expects $n + 3$ arguments which are respectively the SECRO's initial state s_0 , the resulting state, the n arguments passed to the update operation o and finally the return value of o . The postcondition returns a boolean indicating whether to approve or reject the resulting state.

Operation History The definition of a SECRO introduces the concept of an operation history h . The operation history maintains the sequence of mutations that were successfully applied, i.e. the sequence of operations that were approved by both, the associated pre and postcondition.

Commit As mutations are added to the operation history, a replica's history grows over time. To avoid unbounded growth of the operation history, SECROs introduce a `commit` operation. The commit operation commits the current operation history, lifting the replica to a new version $v_i \rightarrow v_{i+1}$. To this end, the replica's version number is incremented, the initial state is replaced by the current state ($s_0 \leftarrow s_i$) and the operation history is discarded, yielding a blank history: $h \leftarrow '()$.

Notice that preconditions are less expressive than postconditions. However, preconditions avoid unnecessary computations by rejecting invalid states prior to the operation's execution. Furthermore, preconditions prevent operations from running on a corrupted state, thus improving the system's robustness.

3.2 Replication and Consistency

Whereas the previous section focused on the formal definition of strong eventually consistent replicated objects (SECROs), we now turn our attention to replication and the consistency model of SECROs.

SECROs embed the strong eventual consistency model into an optimistic replication protocol. Similarly to op-based CRDTs, update operations propagate between the replicas. For the sake of simplicity we assume a causal order broadcasting mechanism without loss of generality, i.e. a communication

medium in which messages arrive in an order that is consistent with the causal happened-before relation. If the underlying communication mechanism does not guarantee causal order broadcasting, one can resort to manually tracking causal dependencies, as explained in Section 2.5.2.

Even though we rely on causal order broadcasting, concurrent operations arrive in arbitrary orders at the replicas. In contrast to CRDTs we do not impose concurrent operations to commute. Therefore, replicas must agree on the same ordering of operations in order to remain eventually consistent. Our goal is to pick an ordering of the operations that is in accordance with the application-dependent semantics. We refer to such an ordering as being a “valid execution”.

Definition 1 *A valid execution is an ordering of the operation history m_1, \dots, m_n in which no pre- or postcondition is violated:*
 $\forall m_i \in h : p_i \wedge a_i$ where $1 \leq i \leq n$, $n = |h|$ and $m_i = (o_i, p_i, a_i)$ is a mutation.

At any moment in time a replica may receive an operation that is concurrent with one or more other operations. Since operations do not commute, receiving an operation requires the replica to re-organize its operation history such that it forms a valid execution. Re-ordering the history is a *local* operation which does not involve network communication. This results from the fact that replicas are deep copied. Hence, each replica carries the preconditions and postconditions that are needed to re-order the history. Notice that re-organizing a replica’s history yields a different state. Therefore, the replica’s current state s_i is updated every time the history changes.

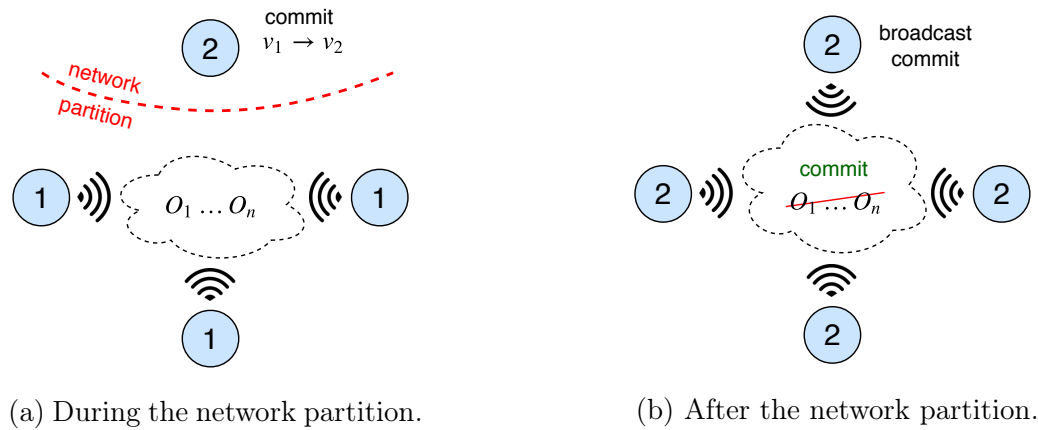


Figure 3.1: Committing a replica in the face of a network partition.

In the previous section we introduced the commit operation which seals the operation history and lifts the replica to a new version ($v_i \rightarrow v_{i+1}$). Since

commit operations are also broadcasted, all replicas seal their old version and transition to the committed version. If we later receive an operation that applies to an old version, the operation is ignored. Therefore, commit should be used with care.

In Figure 3.1a the upper replica commits while suffering from a network partition. Therefore, the upper replica resides in version 2 while the other replicas are still in version 1. Meanwhile, the lower replicas perform a number of operations, O_1 till O_n . Due to the network partition, the operations are concurrent with the commit. Therefore, when the partition fades away (Figure 3.1b), the commit is broadcasted and all replicas transition to version 2, thereby discarding operations $O_1 \dots O_n$.

If programmers use SECROs and associate preconditions and postconditions to the operations, the framework will take care of the following aspects:

1. Replicas converge towards the same valid execution (i.e. eventual consistency).
2. Replicas that received the same updates have identical operation histories (i.e. strong convergence).
3. Replicas eventually take on a valid execution if one exists, or issue an error if none exists (safety and liveness guarantee).

Notice that SECROs cannot guarantee that a valid execution exists. This is because preconditions and postconditions determine which sequences of operations are valid executions. Hence, it is the programmer's responsibility to ensure that a valid execution exists. We define the set of all valid executions, denoted H , as follows:

$$H = \{h \mid \forall m_i \in h : p_i \wedge a_i\} \text{ where } m_i = (o_i, p_i, a_i) \text{ is a mutation}$$

H consists of all re-orderings of the operation history that are valid according to the application-dependent semantics. We assume that for every set of concurrent operations some ordering of the operations is valid. As a result, programs that yield no valid execution - i.e. $H = \emptyset$ - are inherently wrong. Examples of faulty programs include programs with contradicting invariants.

Definition 2 *A faulty program is a program for which no valid execution exists: $\nexists h : \forall m_i \in h : p_i \wedge a_i$ where $m_i = (o_i, p_i, a_i)$ is a mutation.*

To conclude, SECROs provide programmers with a means to define replicated objects that are tailored to the needs of the application and guarantee strong eventual consistency. By specifying invariants programmers implicitly

define custom concurrent behaviours. Hence, SECROs are general-purpose objects that guarantee strong eventual consistency and an outcome that is in accordance with the application-dependent semantics.

3.3 Examples

We now illustrate the usage of SECROs through two examples. The first example implements a replicated linked list allowing insertions at arbitrary positions as well as deletions. The second example is a variation on the first example to implement an ordered linked list. Since both examples build on a traditional linked list, we first discuss the underlying linked list in Section 3.3.1.

For the sake of simplicity both examples assume that the lists contain no duplicate values. As such, we can easily refer to relative positions by referring to a certain element. This assumption does not imply a loss of generality as we can always identify list elements using unique IDs and refer to relative positions using those IDs.

3.3.1 Linked List

The examples presented throughout this chapter build on a traditional implementation of a linked list. We assume that the linked list supports the public interface shown in Figure 3.2.

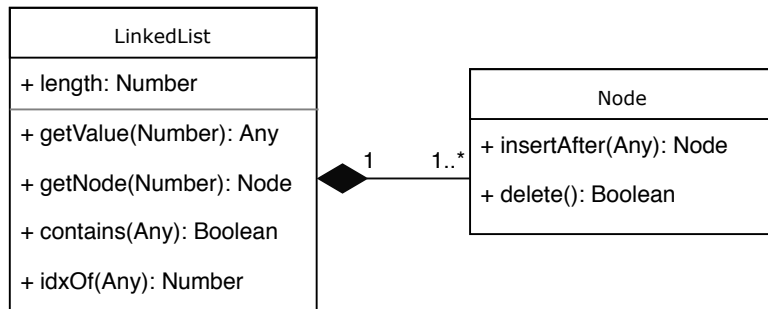


Figure 3.2: UML class diagram of the linked list.

Figure 3.2 depicts two classes, `LinkedList` and `Node`. Conceptually, a linked list is a sequence of nodes where nodes store a value. The linked list defines four methods: `getNode`, `getValue`, `contains` and `idxOf`. The `getNode` method returns the node at a certain index in the list. Lists are indexed starting from 1 such that position 0 refers to the head of the list (i.e. the position before the first element). As such, elements can be prepended

to the list by inserting them after the head of the list. Similarly, `getValue` returns the value at a certain position in the list. Additionally, `contains` and `indexOf` serve to check whether the list contains a certain value, respectively, compute the index of a certain element. When the element does not occur in the list, `indexOf` returns 0, i.e. the index of the head of the list.

Besides navigating through the list, programmers can also manipulate the list. To this end, we first fetch a node using the list's `getNode` method. Afterwards, we use the node's `insertAfter` method to insert an element after the node, or its `delete` method to delete the node.

3.3.2 Replicated Linked List

This example illustrates the implementation of a replicated linked list that supports insertions and deletions, and behaves correctly in the face of concurrent operations. In order to define “correct behaviour” we explicitly capture the programmer's expectations:

`insertAfter(element, value)` This operation inserts the given value after a certain element in the list. Hence, we expect that after executing the operation, the value effectively appears behind the element.

`delete(value)` This operation deletes a certain value from the list. Hence, we expect that after deleting a value, the value no longer occurs in the list.

In order to turn the traditional linked list (see Section 3.3.1) into a replicated linked list that meets the above expectations, we define the following SECRO:

$$(v_1, s_0, s_i, \{getAt\}, \{insertAfter, delete\}, h)$$

Initially, both the replica's initial state s_0 and the current state s_i contain an empty linked list, $s_0 = s_i = '()$. Furthermore, the replica defines a query operation `getAt` and two mutators `insertAfter` and `delete`. Additionally, each replica maintains an operation history h which initially is empty. The `getAt` query fetches an element at a certain index, whereas the `insertAfter` and `delete` mutators perform insertions and deletions respectively. The mutators (i.e. `insertAfter` and `delete`) are extended with state validators which translate the programmer's expectations into invariants.

Throughout this chapter we extensively rely on pseudocode. Therefore, Table 3.1 clarifies the adopted notation. Notice that the underlying implementation ensures that `self` refers to the replica's current state s_i . This state includes one field named `list`, which contains the linked list from Section 3.3.1.

Syntax	Description
<code>def name(arguments):</code>	method definition
<code>var <- val</code>	variable assignment
<code>for val in col:</code>	traversing a collection
<code>#</code>	comments

Table 3.1: Overview of the adopted pseudocode syntax.

The getAt Query Operation

Listing 3.1 shows the `getAt` query operation which returns the value at a certain index in the list. To this end, Line 2 delegates the call to the `getValue` method (see Figure 3.2) of the underlying list.

```
1 def getAt(idx):
2   return self.list.getValue(idx)
```

Listing 3.1: Pseudocode implementation of the `getAt` query operation.

The insertAfter Mutator

The `insertAfter` mutator is a triple:

$$\text{insertAfter} = (\text{insert}, \text{preInsert}, \text{postInsert})$$

`insert` is the actual insertion operation whereas `preInsert` and `postInsert` are the operation's associated precondition and postcondition.

```
1 def insert(element, value):
2   idx <- self.list.idxOf(element)
3   node <- self.list.getNode(idx)
4   return node.insertAfter(value)
```

Listing 3.2: Pseudocode implementation of the `insert` operation.

Listing 3.2 contains the `insert` operation. This operation relies on the API of the underlying linked list to insert the element. First, the operation determines the index of the element after which to insert `value` (Line 2). Afterwards, Lines 3 and 4 fetch the actual node and insert the value after that node.

In order to ensure correct behaviour of the replicated linked list in the face of concurrent updates, we associate a precondition and a postcondition to the insertion operation. Recall from Section 3.1 that preconditions take $n + 1$ arguments, namely the state prior to the execution of the associated

39CHAPTER 3. Strong Eventually Consistent Replicated Objects

operation (which we will refer to as s_B), followed by the n arguments received by that operation. On the other hand, postconditions take two more arguments, being the resulting state (which we denote s_A) as well as the operation's return value¹.

```
1 def preInsert( $s_B$ , element, value):  
2     return element == nil ||  
3          $s_B$ .list.contains(element)
```

Listing 3.3: Precondition for insertions.

Listing 3.3 defines the precondition for insertion operations. This precondition states that the element after which to insert the value must exist (Line 3). A corner case arises when prepending values to the list, which is indicated by `nil`. Therefore, the precondition performs an additional check on Line 2.

```
1 def postInsert( $s_B$ ,  $s_A$ , elem, val, res):  
2     return  $s_A$ .list.idx_of(elem) <  
3          $s_A$ .list.idx_of(val)
```

Listing 3.4: Postcondition for insertions.

Finally, Listing 3.4 defines the postcondition. The postcondition is almost a literal translation of the aforementioned expectations, namely that the inserted element `val` must occur behind the original element `elem`.

A subtlety arises from the fact that users may concurrently insert different values at the same position in the list. As a result, only one of the values can occur at that position. Therefore, the postcondition does not enforce the value to occur right after the element, but rather behind the element.

The delete Mutator

The delete mutator is a triple:

$$delete = (del, /, postDel)$$

The `del` operation is the actual delete operation. Notice that this mutator does not define a precondition.

```
1 def del(val):  
2     if self.list.contains(val):  
3         idx <- self.list.idxOf(val)  
4         self.list.getNode(idx)  
5             .delete()
```

Listing 3.5: Pseudocode implementation of the delete operation.

¹ s_B denotes the state Before the operation's execution, whereas s_A denote the state After the execution.

Listing 3.5 shows the `del` operation. First, the operation checks whether the value occurs in the list (Line 2). If the value occurs in the list, the operation computes the index (Line 3), fetches the node using this index (Line 4) and deletes the node (Line 5).

In contrast to the `insertAfter` mutator, we do not define a precondition for deletions. Intuitively, one might design a precondition to ensure that the value exists. However, if peers concurrently delete the same element only the first deletion observes the element, in which case the others would fail.

```
1 def postDel(sB, sA, val, res):
2   return !sA.list.contains(val)
```

Listing 3.6: Postcondition for deletions.

Finally, Listing 3.6 defines the postcondition for deletions. The postcondition states that the deleted element `val` may not occur in the resulting list (Line 2).

Conclusion

By defining three state validators (`preInsert`, `postInsert` and `postDel`) on top of the sequential linked list implementation, we are able to transform the list into a replicated linked list that behaves as expected.

Conceptually, the state validators define an implicit conflict resolution strategy. In the face of conflicting insertions and deletions, the invariants prescribe an ordering of the operations in which the insertions precede the deletions, and as such solve the conflicts.

Finally, this example showcased a major advantage of SECROs over CRDTs. Whereas various implementations of a linked list CRDT can be found in the literature (Letia, Prego, & Shapiro, 2009; Roh, Jeon, Kim, & Lee, 2011), all of them resort to tombstones to mark elements as deleted. Hence, CRDTs simulate deletions whereas SECROs support true deletions by re-organizing the operation history.

3.3.3 Ordered Linked List

We now modify the replicated linked list from the previous example with different concurrent behaviour. Assuming peers perform sorted insertions (using `insertAfter`), the resulting list must be sorted. However, when two or more peers concurrently insert an element at the same position in the list, the replicas agree on an arbitrary ordering of the insertions. Our goal is to modify this behaviour such that concurrent insertions result in an ascending order of the elements.

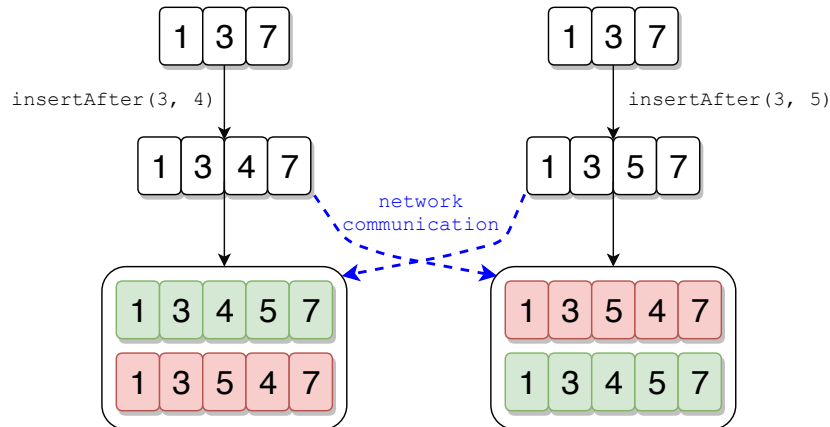


Figure 3.3: Concurrent list insertions at the same position.

To clarify the goal of this example Figure 3.3 depicts the interactions between two peers, Alice (left) and Bob (right). Initially, both start with a replicated linked list '(1, 3, 7). Alice issues `insertAfter(3, 4)` while concurrently Bob issues `insertAfter(3, 5)`. When exchanging the operations, both peers notice that the operations are concurrent. Since concurrent operations have no predefined order, the resulting list will be '(1, 3, 5, 4, 7) or '(1, 3, 4, 5, 7) depending on the IDs of the operations. In this example we want to force the concurrent insertions to yield the sorted order: '(1, 3, 4, 5, 7).

In order to achieve the desired behaviour, we only need to modify the `postInsert` postcondition from the previous example (Listing 3.4).

```

1 def postInsert( $s_B$ ,  $s_A$ , elem, val, res):
2   idx <-  $s_A$ .list.idx_of(val)
3   prev <-  $-\infty$ 
4   next <-  $+\infty$ 
5   if idx > 1:
6     # Not the first element
7     prev <-  $s_A$ .list.getValue(idx-1)
8   if idx <  $s_A$ .list.length:
9     # Not the last element
10    next <-  $s_A$ .list.getValue(idx+1)
11  return prev < val && val < next

```

Listing 3.7: Postcondition for ordered concurrent insertions.

Listing 3.7 contains the modified postcondition. First, the postcondition computes the index of the inserted element `val` (Line 2) and assumes that the preceding and succeeding values are $-\infty$ and $+\infty$ respectively (Lines 3 to 4). Afterwards, the postcondition fetches the actual values that precede and succeed the element (Lines 5 to 7 and 8 to 10) and enforces the previous value to be smaller and the next value to be larger (Line 11). Since the first and last elements do not have a preceding, respectively, a succeeding value,

Lines 5 and 8 perform some extra checks. Recall that the first element starts at position 1.

The described postcondition implicitly forces concurrent insertions to execute in ascending order. This yields a single valid execution².

To conclude, this example showcased the general applicability of SECROs. Whereas JSON CRDTs hardcode specific concurrent behaviour in the merge procedures, SECROs provide state validators as a mechanism to effectively customize concurrent behaviour. We emphasize the fact that JSON CRDTs cannot reproduce this example since they hardcode different concurrent behaviour.

3.4 Implementation

This section showcases a partial pseudocode implementation of the strong eventually consistent replicated data type. First, Section 3.4.1 depicts the evaluation of query operations. Afterwards, Section 3.4.2 focuses on applying updates, paying special attention to the use of state validators to guarantee correctness and strong eventual consistency. Finally, Section 3.4.3 reviews the implementation of the commit operation.

Remember from Section 3.1 that every replica maintains a version number and two states, being the initial state s_0 and the current state s_i . Therefore, replicas are objects containing four fields: `version`, `initial_state`, `state` and `history`. The `initial_state` field corresponds to s_0 whereas the `state` field corresponds to s_i . Notice that `self` refers to the replica object itself.

Additionally, this section introduces specific notation: $<_h$ refers to the causal “happened-before” relation and $|_c$ denotes concurrent operations. When an operation occurs in the operation history of a replica, we say that the operation is delivered at the replica. Hence, operation A happened before B ($A <_h B$) if operation A was delivered before executing B . On the other hand, two operations are concurrent if neither one happened before the other (Lamport, 1978), i.e. $A |_c B \iff A \not<_h B \wedge B \not<_h A$.

3.4.1 Applying Queries

Listing 3.8 shows how query operations are evaluated. Since query operations are side effect free, it is safe to apply them directly on the replica’s internal state. Hence, the code fetches the replica’s current state (Line 2) and applies the query on that state (Line 3). This yields the query’s return value.

²We assumed a list without duplicates.


```

1 def apply_query(replica, query):
2   state <- self.state
3   return apply(state, query)

```

Listing 3.8: Applying a query operation on a replica.

3.4.2 Applying Updates

Before delving into the implementation of update operations, we clarify the notion of a mutator. In contrast to Section 3.1, we use the term “mutator” or “update” to refer to the actual update operation o instead of the triple (o, p, a) .

```

1 def issue_update(operation):
2   op <- make_op(operation)
3   result <- apply_update(op)
4   broadcast(op)
5   return result
6
7 def make_op(operation)
8   clock <- logical_timestamp()
9   uid <- generate_unique_id()
10  return (clock, operation, uid)

```

Listing 3.9: Issuing an update operation.

Whenever a replica experiences an update, the replica invokes `issue_update`. As shown in Listing 3.9, `issue_update` generates a description of the update (Line 2). This description includes the operation, a logical timestamp and a unique ID identifying the update (Lines 7 to 10). Afterwards, `issue_update` relies on `apply_update` to perform the actual update (Line 3) and broadcasts the operation to the other replicas (Line 4). Finally, `issue_update` returns the operation’s result value on Line 5.

Listing 3.10 shows the `receive_update` operation which handles updates sent by other replicas. Upon receiving a remote operation, `receive_update` delegates the operation to `apply_update` (Line 2). Notice that the received operation is not broadcasted.

```

1 def receive_update(op):
2   return apply_update(op)

```

Listing 3.10: Receiving updates.

Remember from Section 3.2 that updates do not necessarily commute. Therefore, replicas maintain an operation history that is organized to guarantee strong eventual consistency and “correct” behaviour. The key is to design a deterministic algorithm such that replicas that received the same updates pick the same valid execution.

However, operations are timestamped with logical clocks, and, therefore, only exhibit a partial order (see Section 2.1). In other words, concurrent operations are incomparable. Hence, if two clocks (C_1 and C_2) are concurrent $C_1 |_c C_2$, then neither $C_1 <_h C_2$, $C_2 <_h C_1$, nor is $C_1 = C_2$. As a result, a set of n concurrent operations has $n!$ potential orderings, namely all permutations.

To avoid the complexities that arise from partial orders, we design the operation history to exhibit a *total* order defined by $<_h$. To this end, the operation history is a sequence of groups, where each group is a set of concurrent operations (i.e. unordered). We order the groups according to the following total order:

$$G_1 <_h G_2 \iff \forall O \in G_1 : \forall O' \in G_2 : O <_h O' \quad (3.1)$$

Hence, a group G_1 happened before a group G_2 if every operation in G_1 causally precedes the operations from G_2 . Recall that two operations are concurrent if neither one happened before the other. Similarly, we define two groups to be concurrent if neither group happened before the other, i.e. $G_1 |_c G_2 \iff G_1 \not<_h G_2 \wedge G_2 \not<_h G_1$. Therefore, as soon as an operation O_1 of group G_1 is concurrent with an operation O_2 from G_2 , both groups are concurrent since $O_1 |_c O_2$ implies that $G_1 \not<_h G_2$ and $G_2 \not<_h G_1$. For the same reason we define an operation O to be concurrent with a group of operations G , if the operation is concurrent with at least one operation from the group:

$$O |_c G \iff \exists O' \in G : O |_c O' \quad (3.2)$$

Hence, applying an update operation O requires inserting the operation in the right group(s) of the operation history, according to the above total order. Afterwards, the replica can enumerate all potential executions by permuting the group to which it added the operation. Finally, the replica uses the application-dependent invariants (i.e. the pre and postconditions) to find a valid execution among the set of permutations. Once a valid execution is found, this ordering becomes the replica's new history. This approach lies at the basis of the `apply_update` function shown in Listing 3.11.

```

1 def apply_update(op):
2   history <- self.history
3   concurrent_groups <- get_concurrent_groups(op, history)
4   if concurrent_groups is empty:
5     history <- insert_group('(op)', history)
6     (valid, result, state) <- is_valid(history, op)
7     if valid:
8       self.history <- history
9       self.state <- state
10    return result
11  else:
12    group <- merge_groups(concurrent_groups)
13    group <- cons(op, group)

```

```

14   group <- sort(group)
15   permutations <- permute(group) # lazy!
16   for permutation in permutations:
17     if respects_causality(permutation):
18       potential_history <- replace_groups_by(history, concurrent_groups,
19                                             group)
19       (valid, result, state) <- is_valid(potential_history, op)
20       if valid:
21         self.history <- potential_history
22         self.state <- state
23         return result
24   throw faulty_program_exception

```

Listing 3.11: Applying an update operation.

The implementation of `apply_update` listed above, starts by fetching all groups that are concurrent with the update operation (Line 3). Two possibilities arise, either the operation is concurrent with no groups (Lines 4 to 10) or the operation is concurrent with one or more groups (Lines 11 to 23).

In the former case, the operation forms a singleton group (op) and `insert_group` returns a new history in which the group is inserted at its correct position (Lines 4 to 5), according to the $<_h$ relation (Equation (3.1)). Afterwards, Line 6 checks the validity of the extended operation history using the `is_valid` operation (see Listing 3.12). If the history is approved, the code updates the replica's history and current state, and returns the operation's result (Lines 7 to 10). Otherwise, the code jumps to Line 24 and raises an exception since the application's invariants cannot be met.

In the latter case the operation is concurrent with one or more groups. Since the operation makes the groups concurrent, the groups are merged and the operation is added to the merged group (Lines 12 and 13). Line 14 sorts the resulting group according to the following total order:

$$O_1 < O_2 \iff (C_1 <_h C_2) \vee (C_1 |_c C_2 \wedge p_1 < p_2)$$

where $O_i = (C_i, p_i)$ is an operation
 $C_i =$ a logical timestamp
 $p_i =$ the operation's unique ID

Since replicas receive concurrent operations in arbitrary orders, sorting the group enforces all replicas to start from the same ordering. As such, the replicas generate permutations deterministically, i.e. in the same order, on Line 15. For the sake of performance, permutations are generated lazily.

After sorting the group, `apply_update` loops through the permutations searching for the first valid execution (Lines 16 to 23). From Equation (3.2) it follows that some operations within the merged group may still be causally

related. Therefore, Line 17 discards permutations that violate causality. For permutations that respect causality, the code constructs a potential history by replacing the concurrent groups by the merged permuted group (Line 18). To this end, the `replace_groups_by` function returns a new history in which the permuted group replaces the first concurrent group and the remaining concurrent groups are deleted. Afterwards, the `is_valid` function is used to tentatively execute the potential history similarly as before (Lines 19 to 23). However, if the potential history is rejected, the code jumps back to Line 16 and generates the next permutation. Finally, if all permutations are rejected, Line 24 throws an exception since the application's invariants cannot be met.

Finally, remains to discuss the implementation of `is_valid` which relies on the application's invariants to approve or reject a history. The idea is to tentatively execute the updates from the history, using the preconditions and postconditions to validate each intermediate state. If an update is rejected by one of its invariants, then the entire operation history is corrupted and the history is rejected.

```

1 def is_valid(history, newOp):
2     state_dict <- new Map()
3     state <- copy(self.initial_state)
4     result <- nil
5     for group in history:
6         for operation in group:
7             pre <- get_precondition(operation)
8             state_dict.set(operation.id, copy(state))
9             if pre(state):
10                res <- apply(state, operation)
11                if operation == newOp:
12                    result <- res
13            else:
14                return (false, nil, nil)
15        for operation in group:
16            post <- get_postcondition(operation)
17            originalState <- state_dict.get(operation.id)
18            if !post(originalState, state):
19                return (false, nil, nil)
20    # All pre- and postconditions succeeded
21    return (true, result, state)

```

Listing 3.12: Validating an operation history.

Listing 3.12 contains the `is_valid` function. Recall that updates may not alter the replica's initial state. Instead, updates are added to the operation history. For this reason, `is_valid` copies the replica's initial state (Line 3) before tentatively executing the history's operations.

Lines 5 to 19 loop through the groups of the operation history. Each group is traversed twice, once to assert the preconditions and apply the updates (Lines 6 to 14) and once to assert the postconditions (Lines 15 to 19). When `is_valid` first loops over the operations of the group, it fetches the operation's precondition (Line 7) and uses this precondition to validate the current

state (Line 9). If the precondition holds `is_valid` applies the update, otherwise it returns a triple (`false`, `nil`, `nil`) indicating the failure (Lines 10 to 14). Notice that the code also stores a copy of the state prior to the execution of the precondition (Line 8) because this state will be passed as an argument to the postcondition. The second time `is_valid` loops through the group, it validates the state that results from this group of concurrent operations. Therefore, `is_valid` fetches each operation's postcondition (Line 16) and calls the postcondition with the original state (Line 17) and the resulting state (Line 18). If a postcondition fails, then by Definition 1 the history is not a valid execution and `is_valid` returns a failure status (Line 19). Finally, if all preconditions and postconditions succeed, the code reaches Line 21 and returns the resulting state as well as the operation's return value.

3.4.3 The Commit Operation

As explained in Section 3.1, `commit` seals the operation history of a replica, thereby lifting the replica to a new version. This means that the replica's initial state is updated and the replica starts off with a blank history. As such, `commit` avoids the operation history to grow unbounded. However, operations that are concurrent with the `commit` are discarded, since those operations belong to the previous version (see Figure 3.1).

Similarly to regular operations, replicas may issue commits concurrently. Therefore, we design `commit` operations to commute. This ensures that replicas converge in the face of concurrent commits.

```

1 def issue_commit():
2   commit_state <- self.state
3   op <- make_op("commit", commit_state)
4   apply_commit(op)
5   broadcast(op)
6
7 def make_op(operation, state = nil):
8   version <- self.version
9   clock <- logical_timestamp()
10  uid <- generate_unique_id()
11  return (state, version, clock, operation, uid)

```

Listing 3.13: Issuing a commit operation.

Whenever a replica experiences a local commit, the replica invokes `issue_commit` which is shown in Listing 3.13. This function creates a description of the commit operation (Line 3). `make_op` is extended (cf. Listing 3.9) as to include the replica's version number and current state in the description (Lines 7, 8 and 11). The latter is needed since replicas are bound to use the committed state. After generating the description, `issue_commit` delegates the call to `apply_commit` (Line 4) and broadcasts the commit operation

(Line 5).

Listing 3.14 shows the `receive_commit` function which handles commits sent by other replicas. Upon receiving a remote commit operation, `receive_commit` delegates the operation to `apply_commit` (Line 2).

```
1 def receive_commit(commit):
2   apply_commit(commit)
```

Listing 3.14: Receiving a commit operation.

We now turn our attention towards the commutativity design of commit. When two or more replicas concurrently issue a commit, the commit operations are exchanged and eventually the replicas notice the conflict. To break ties consistently, replicas elect the commit with the smallest ID, thereby discarding the others. As such, the order in which commits are received is immaterial, since replicas systematically pick the commit with the smallest ID.

```
1 def apply_commit(description):
2   # destructure the description
3   (state, version, clock, op, id) <- description
4   if version == self.version:
5     update_version(state, id, true)
6   else if version == self.version - 1:
7     if id < latest_commit:
8       update_version(state, id, false)
9
10  def update_version(state, commit_id, increment_version):
11    self.initial_state <- state
12    self.state <- state
13    self.history <- '()'
14    self.latest_commit <- commit_id
15    if increment_version:
16      self.version <- self.version + 1
```

Listing 3.15: Committing a replica.

Listing 3.15 shows the `apply_commit` function which distinguishes between three cases: either the commit operation commits the current state, the previous state or an elder state. In the first case, the committed version matches the replica's current version (Line 4). Hence, `apply_commit` calls `update_version` (Line 5) which increments the replica's version number, replaces the replica's initial and current states by the committed state, clears the operation history and stores the ID of this commit operation (Lines 11 to 16).

In the second case, the version number matches the replica's previous version (Line 6). This means that the commit operation applies to the previous version v_{i-1} . As a result, this commit operation (say c_2) is concurrent with the commit (c_1) that caused the replica to update its version: $v_{i-1} \rightarrow v_i$. This follows from the fact that if $c_1 <_h c_2$ then c_2 is aware of the newer

version v_i and c_2 would commit version v_i instead of v_{i-1} . To ensure convergence, replicas retain the commit operation with the smallest ID. Therefore, if this commit operation has a smaller ID than the latest commit (Line 7), the replica calls `update_version` to overwrite its state by the committed state, clear the history and store the ID of this commit operation (Lines 11 to 14). The replica's version number is not incremented, since the concurrent commit already did.

Finally, commits that match elder versions are ignored because they are lagging behind. This cannot happen if a causal order broadcasting mechanism is used.

```

1 def receive_update(op):
2   # Destructure the operation
3   (state, version, clock, operation, id) <- op
4   if version == self.version:
5     return apply_update(op)

```

Listing 3.16: Ignoring operations that apply to elder versions.

Similarly, operations that apply to previous versions of a replica are ignored. To this end, Listing 3.16 extends the `receive_update` function (cf. Listing 3.10) with an extra check (Line 4).

3.4.4 Time Complexity Analysis

We conclude this section about the implementation of SECROs, with a brief analysis of its time complexity. First, we analyze the time complexity of query operations. Afterwards, we analyze the time complexity of mutations.

Since query operations are side-effect free they are directly applied on the replica's current state s_i (see Listing 3.8). As a result, the time complexity of `apply_query` depends on the query operation at hand, $O(\text{query})$.

$$O(\text{is_valid}) = O(|s|) + O(n \cdot (\text{pre} + \text{op} + \text{post})) \quad (3.3)$$

$$= O(|s| + n \cdot b) \quad (3.4)$$

$$O(\text{apply_update}) = O(n!) \cdot O(\text{is_valid}) \quad (3.5)$$

$$= O(n!) \cdot O(|s| + n \cdot b) \quad (3.6)$$

$$= O(n! \cdot |s| + n! \cdot n \cdot b) \quad (3.7)$$

Regarding the execution of updates (`apply_update` in Listing 3.11), worst case all n operations are concurrent. As a result, the history consists of a single group which contains all update operations. Afterwards, when looping through the permutations of this group (Line 16), this potentially results in $n!$ iterations. Regarding the loop's body, `respects_causality` and `replace_groups_by` can be implemented in $O(n)$. However, validation of

the history is at best linear, i.e. $\Omega(n)$. Therefore `is_valid` dominates the loop's execution time, yielding Equation (3.5).

Equation (3.3) presents the running time of `is_valid`. First, we copy the replica's internal state: $O(|s|)$, on Line 3 in Listing 3.12. Afterwards, we loop through the operations of each group in the history (Lines 5 and 6), which is in $O(n \cdot \text{body})$. For each operation we execute its precondition, the operation and its postcondition (Lines 9, 10 and 18), yielding $O(\text{body}) = O(\text{pre} + \text{op} + \text{post})$. Equation (3.4) rewrites the body's execution time as b .

Finally, Equation (3.6) rewrites `is_valid`'s time complexity. We then use the distributive property to merge the different terms, which yields the final time complexity of update operations, shown in Equation (3.7).

We conclude that the time complexity of update operations is bound to the size of the replica's internal state, the number of update operations since the latest commit and the performance of the operations and their associated state validators.

3.5 Insights

We now outline a number of advanced insights. First of all, commutative operations are conflict-free, hence, they do not require preconditions or postconditions. Note, however, that those operations must be commutative with every other operation, including itself.

Regarding performance, Section 3.4.4 revealed an $O(n! \cdot |s| + n! \cdot n \cdot b)$ time complexity for update operations. Although this time complexity is bad, we argue that in practice this is often acceptable. For the sake of clarity, we distinguish between the total number of operations n , and the number of concurrent operations c contained by the group we are re-organizing (Line 15 in Listing 3.11). Therefore, we rewrite the time complexity as $O(c! \cdot |s| + c! \cdot n \cdot b)$, where in the worst case all operations are concurrent $c = n$.

Assuming reasonable network connectivity, operations are exchanged within relatively small time spans. Hence, peers that are offline will soon regain connectivity and broadcast their operations. Therefore, most replicas experience acceptable amounts of concurrent operations, yielding $c \ll n$.

However, even if c is small enough for $c!$ to be neglected, the time complexity remains $O(|s| + n \cdot b)$, where b depends on the application's operations, preconditions and postconditions. Therefore, if operations are linear ($b = n$), the complexity becomes quadratic: $O(|s| + n^2)$.

To cope with the aforementioned problem, practical systems must periodically commit replicas. As such, the operation history remains small enough for the performance to be acceptable. Committing replicas is key to achieve

decent performance.

To get a better understanding, assume a commit strategy in which replicas are committed every X operations. As a result of this strategy replicas support no more than X concurrent operations. Hence, for a commit interval of $X = 1$ we get the best performance, but give up on concurrency. On the other hand, if replicas are never committed ($X = \infty$) they can in theory support an infinite amount of concurrent operations, at the cost of performance.

Commit thus entails a *trade-off* between performance and concurrency. Frequently committing replicas leads to better performance but less concurrency. On the other hand, occasionally committing replicas leads to more concurrency at the cost of performance. Hence, the rate at which replicas are committed is application-specific.

3.6 Conclusion

Up till now conflict-free replicated data types (CRDTs) were the only mechanism for implementing strong eventual consistency. However, CRDTs require operations to commute which affects their general applicability and poses major challenges for their integration into real-world distributed systems.

To address the aforementioned problems, we proposed strong eventually consistent replicated objects (SECROs), a data type that guarantees SEC without restrictions on the operations. Programmers express invariants which define the object's behaviour in the face of concurrent operations. As such, arbitrary objects become SEC replicated objects.

In contrast to CRDTs, designing application-specific SECROs requires no craftsmanship in distributed programming. Replicated objects are implemented similarly to their local counterpart, with the addition of preconditions and postconditions to define custom behaviour.

4

CScript

This chapter describes CScript, a distributed programming language that eases the development of both, available and consistent distributed systems. First, Section 4.1 briefly motivates the need for CScript and provides a high-level introduction to the language. Afterwards, Section 4.2 discusses a grocery list application which acts as a motivating example throughout this chapter. Section 4.3 provides an overview of the CScript language. We then introduce services, a central concept of the CScript language, in Section 4.4. Section 4.5 introduces a special type of object, called replica. Finally, Section 4.6 concludes this chapter with a brief discussion of CScript's current limitations.

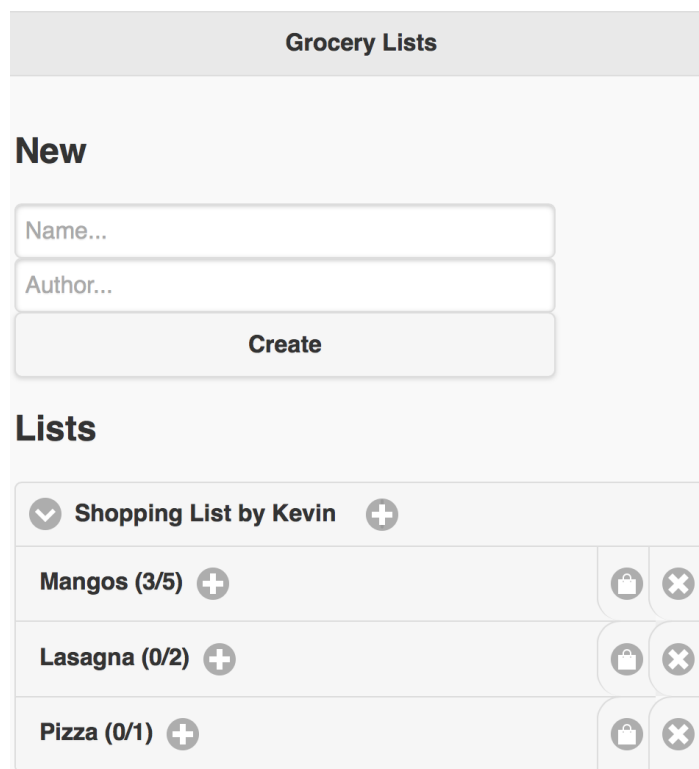
4.1 Introduction

When facing network partitions, real-world distributed systems are forced to choose between consistency (CP) or availability (AP). Although every distributed system faces these complexities, merely a few programming languages include native support to help the programmer with this task (see Table 2.1). To the best of our knowledge, no distributed programming language supports both strong consistency (CP) and strong eventual consistency (i.e. AP with an additional weak consistency guarantee). Since there is a growing need for such languages, we developed the CScript language.

CScript is a domain-specific language for distributed programming which extends JavaScript¹ with language constructs for replication and consistency. CScript embeds native support for strong consistency (CP) and availability (AP). In contrast to other distributed programming languages, CScript offers a wide range of available (AP) constructs. Those constructs support different consistency models, including “no consistency” and “strong eventual consistency” (see Section 2.3).

4.2 Motivating Example

Throughout this chapter we showcase the CScript language using a grocery list application as a motivating example. Therefore, we first describe the functional requirements of the grocery list application. Afterwards, we describe the non-functional requirements as well as the structure of the application.



The screenshot shows a web application titled "Grocery Lists". It features a "New" section with two input fields labeled "Name..." and "Author...", followed by a "Create" button. Below this is a "Lists" section displaying a list of items under the heading "Shopping List by Kevin". The items are "Mangos (3/5)", "Lasagna (0/2)", and "Pizza (0/1)". Each item has a plus sign icon to its right and a shopping bag icon and a close (X) icon to its left.

Figure 4.1: Overview of the grocery list application.

¹CScript runs on top of the NodeJS runtime.

4.2.1 Functional Requirements

Figure 4.1 shows an overview of the grocery list application. The application displays all grocery lists under the “Lists” section. To create a new grocery list, the user must enter a name for the list as well as their nickname, and click on the “Create” button. The newly created list will then appear among the other lists.

Users can add items to a grocery list by clicking the plus button next to the list. The user will be prompted to enter a name and the requested quantity of the item. Afterwards, the item appears in the grocery list. If one needs more pieces of that item, the quantity can be incremented by clicking the plus button next to the item. Notice that every item displays a progress status. For mangos this is $2/3$ which indicates that two out of the three requested mangos were bought.

Additionally, users can buy a certain quantity of an item by clicking the shopping button next to the item. The user will be prompted how many pieces he wants to buy. The user then enters a quantity and confirms his request, whereafter he waits for an answer from the system. The system will either approve or reject the buy request and inform the user of this decision.

Finally, users can delete items from a grocery list by clicking the delete button next to the item. The item then disappears from the list. Note, however, that the item may later reappear if someone else adds the item.

4.2.2 Non-functional Requirements

In addition to the functional requirements, the grocery list application defines the following non-functional requirements:

Automatic Sharing Grocery lists must be shared between all users.

Hence, when a user creates a new grocery list, the other users must automatically see that list on their application.

Consistent Purchases Users should not be able to buy the same item concurrently since buying an item twice is a waste of money. Hence, the system must guarantee purchases to happen consistently. To this end, users must explicitly request approval from the system before buying an item.

Offline Availability At any moment in time users should be able to add, delete or update items of a grocery list, otherwise they may forget about it. This implies that all functionality (except purchases) must be offline available. However, if a user updates a shared grocery list while being

offline, his list becomes different from the others. The system will need to solve this problem when the user comes back online.

4.2.3 Structure of the Application

We now discuss the internal structure of the grocery list application. To this end, we use the application's class diagram, shown in Figure 4.2.

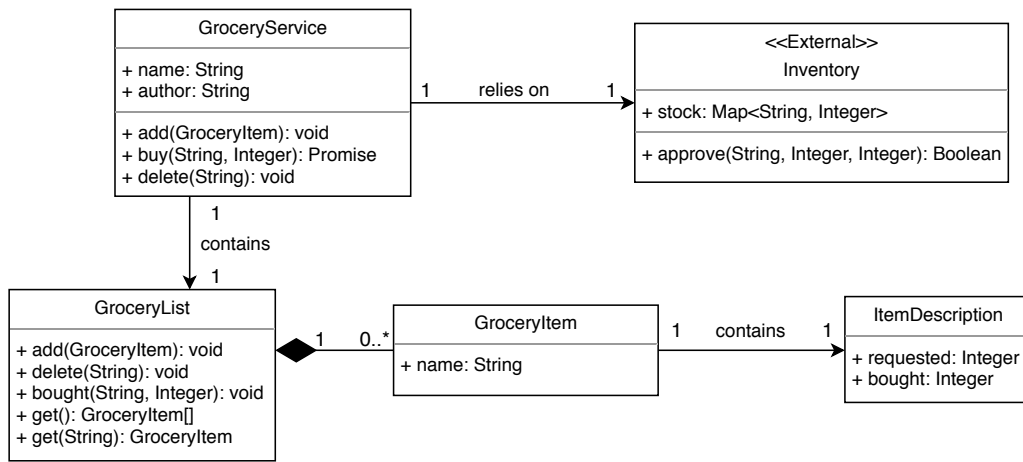


Figure 4.2: UML class diagram of the grocery list application.

A grocery list consists of two components: an available list and a consistent inventory of purchases. These components are bundled into a **GroceryService**. Additionally, grocery services also encapsulate the grocery list's name and author, whose combination uniquely identifies the grocery service.

The items of a grocery list are maintained by a **GroceryList** object. For each item the object keeps an **ItemDescription** containing the requested quantity and the number of purchased pieces. Using the **add** and **delete** methods we can add new items, respectively, delete items from the list. The **bought** method informs the list that a certain quantity of an item was bought. The list then updates the item's description and refreshes the user interface.

Finally, every grocery service relies on a strongly consistent **Inventory**. This inventory is an external entity that maintains an overview of all purchases. Hence, for each item of the grocery list, the inventory stores the number of purchased pieces.

4.3 Language Overview

Before we delve into the different aspects of CScript, we give a high-level overview of the language.

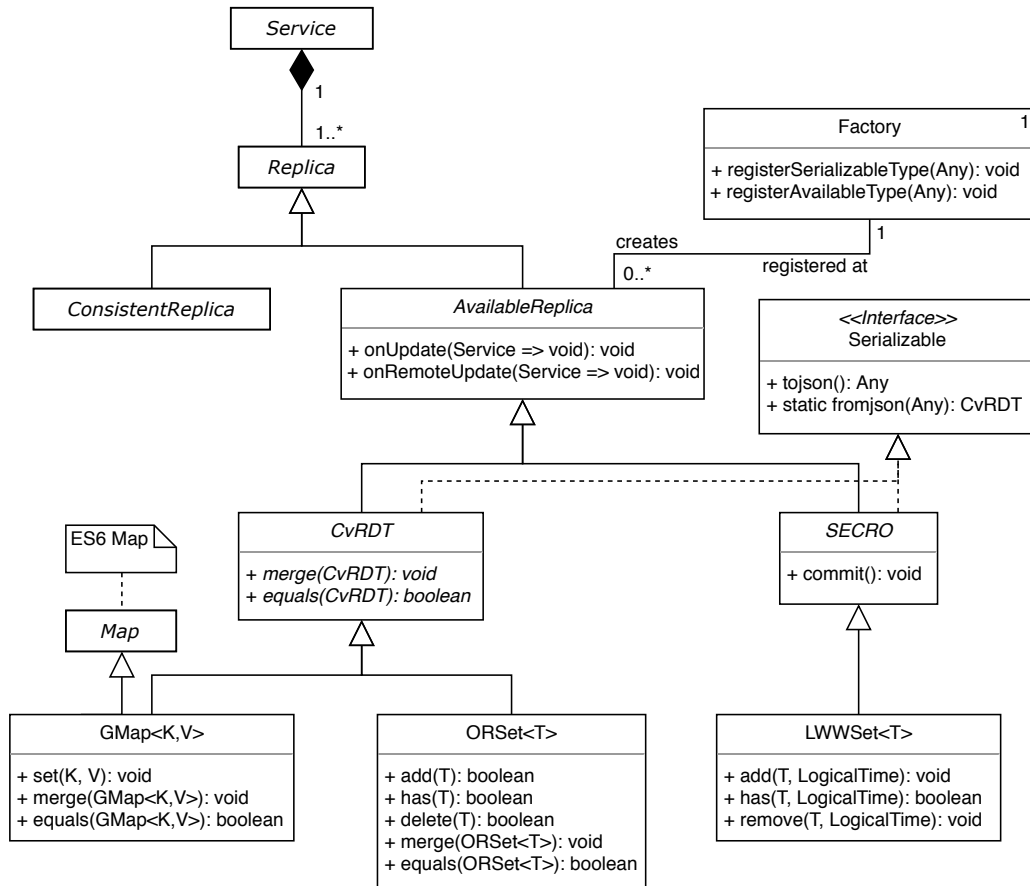


Figure 4.3: UML class diagram of the CScript language.

Figure 4.3 describes CScript in the form of a class diagram. Services form the basic building blocks of CScript. A service is a collection of data objects, which are called replicas. CScript supports two types of replicas, namely available and consistent replicas. These replicas are flexible, fine-grained constructs for implementing available or consistent components. Remarkably, services can be partly consistent and partly available.

The rest of this class diagram will gradually be explained throughout this chapter. Also notice that JavaScript does not support interfaces and abstract classes, however, the class diagram includes them to illustrate the conceptual ideas.

4.4 Services

CScript extends JavaScript with replicated data containers, called services. Each service encapsulates a number of replicas, which are objects with an additional availability or consistency guarantee, and coordinates between the replicas in order to provide certain functionality. CScript also provides a peer-to-peer publish-subscribe mechanism (Eugster, Felber, Guerraoui, & Kermarrec, 2003). This publish-subscribe mechanism allows applications to discover services without knowing their physical address beforehand.

Since services are exchanged over the network, every service must be self-contained. This implies that services may not depend on their lexical scope. Hence, services resemble AmbientTalk’s isolates (see Section 2.7.4).

4.4.1 Defining Services

To illustrate the use of services, Listing 4.1 shows the implementation of the `GroceryService` from our motivating example. Recall from Section 4.2.3 that a `GroceryService` maintains the actual grocery list as well as an inventory.

```
1 service GroceryService {
2   rep groceryList = new GroceryList();
3   rep inventory   = new Inventory();
4
5   constructor(name, author)
6     this.name     = name;
7     this.author  = author;
8   }
9
10  add(item) {
11    return this.groceryList.add(item);
12  }
13
14  delete(itemName) {
15    return this.groceryList.delete(itemName);
16  }
17
18  buy(itemName, buyingQuantity) { /* ... */ }
19 }
```

Listing 4.1: Implementation of the grocery service.

Listing 4.1 defines the `GroceryService` using the `service` keyword (Line 1). Similarly to class definitions in ES6², services have a constructor method which initializes the object (Lines 5 to 8). This grocery service contains two regular fields, namely the grocery list’s name and author (Lines 6 and 7).

²ECMAScript 6

Additionally, this `GroceryService` defines two replicas: `groceryList` and `inventory` (Lines 2 and 3). The former replica is the actual grocery list whereas the latter replica is the inventory.

Finally, the grocery service defines three methods: `add`, `delete` and `buy` (Lines 10 to 18). The `add` and `delete` methods are used to add, respectively, delete grocery items. To this end, both methods delegate the call to the underlying grocery list. The `buy` method is shown in Listing 4.2.

```
1 buy(itemName, buyingQuantity) {
2   return new Promise((resolve, reject) => {
3     const stockQuantity = this.groceryList.get(itemName).bought;
4     this.inventory
5       .then(inventory => {
6         return inventory.approve(itemName, stockQuantity,
7           buyingQuantity)
8       })
9       .then(accepted => {
10        if (accepted) {
11          this.groceryList.bought(itemName, buyingQuantity);
12          resolve();
13        }
14        else {
15          reject("Buy request rejected.");
16        }
17      });
18 }
```

Listing 4.2: Buying a certain quantity of a grocery item.

Recall from Section 4.2.2 that users need explicit approval from the inventory in order to buy an item. Therefore, the `buy` method (Listing 4.2) needs to coordinate between the inventory and the local grocery list. The inventory is an external entity, hence, all communication is asynchronous. As a result, the `buy` method is also asynchronous.

The `buy` method expects two arguments: the name of the item and the quantity the user wants to purchase. The method returns a promise that will be resolved if the buy request is approved and rejected otherwise. First, the method asks its local grocery list how many pieces of this item were already purchased and stores that amount as `stockQuantity` (Line 3). Afterwards, the method sends a buy request to the inventory (Line 6). This request includes the item's name and the quantity the user wants to buy, but also the number of pieces the user believes were already purchased. Hence, informally this buy request translates to: "The user would like to buy X pieces of item A given that he has N pieces of that item at home". In case a concurrent buy request occurs, the user's local view - which is included in the buy request - becomes inconsistent and this conflict is detected by the inventory. The inventory will accept the request only if the user's local view is consistent with the inventory.

Upon receiving an answer from the inventory (Lines 8 to 16) the method checks whether the buy request is accepted or rejected. In case the request is accepted, the method informs the grocery list that the user bought `buyingQuantity` pieces of the item and resolves the promise (Lines 10 and 11). Otherwise, the method indicates the failure by rejecting the promise (Line 14).

4.4.2 Exporting and Discovering Services

To exchange services between applications, CScript has a built-in topic-based publish-subscribe mechanism (Eugster et al., 2003). Programmers export services on the network using the `publish as` construct. This construct expects two arguments, a service and a type tag, and makes the service remotely discoverable under the given type tag.

```
1 deftype Grocery
2 function createGrocery(name, author) {
3   const gservice = new GroceryService(name, author);
4   publish gservice as Grocery;
5   processService(gservice);
6   return gservice;
7 }
```

Listing 4.3: Exporting grocery services on the network.

In our motivating example, each time the user creates a new grocery list the user interface sends a message to the back-end. The back-end then calls the `createGrocery` function with the list's name and author, whose implementation is shown in Listing 4.3. First, the function creates a new grocery service with the received name and author (Line 3). Afterwards on Line 4, the function publishes the newly created service under the `Grocery` type tag. Notice that this type tag was previously defined using the `deftype` construct on Line 1. Finally, the function returns the newly created service on Line 6. For the moment, ignore the `processService` call on Line 5.

To discover published services CScript provides the `subscribe with` construct. This construct expects two arguments: a type tag and a callback. Whenever a service of the given type becomes available, the associated callback is triggered.

```
1 const services = new Map();
2 subscribe Grocery with gservice => {
3   const name = gservice.name,
4     author = gservice.author,
5     id = `${name} by ${author}`;
6   services.set(id, gservice);
7   processService(gservice);
8 }
```

Listing 4.4: Subscribing to grocery services.

Listing 4.4 subscribes to services of the `Grocery` type (Line 2). The provided callback is parametrized with the discovered service. Upon discovering a service, the program fetches the service’s name and author (Lines 3 and 4) and uses this information to create a unique identifier for the service (Line 5). Finally, the program stores the discovered service, on Line 6.

Notice that `name` and `author` are properties of the service. When discovering the service the program gets a copy of those properties. These properties are not kept consistent.

4.5 Replicas

CScript replicas are objects which provide an additional availability or consistency guarantee (see Figure 4.3). The object’s methods define the replica’s public interface. Programmers cannot access a replica’s internal state as this would circumvent the replica’s interface. First, Section 4.5.1 shows how to define replicas. Section 4.5.2 elaborates on consistent replicas and showcases the implementation of the strongly consistent inventory. Afterwards, Section 4.5.3 elaborates on available replicas and shows the implementation of the actual grocery list.

4.5.1 Defining Replicas

Services define replicas using the `rep` keyword, similarly to variable definitions. In contrast to variables, replicas are instance properties of the service. Listing 4.5 shows a fragment from the implementation of the `GroceryService`. On Lines 2 and 3 the service defines two replicas: `groceryList` and `inventory`.

```
1 service GroceryService {  
2     rep groceryList = new GroceryList();  
3     rep inventory   = new Inventory();  
4     // ...  
5 }
```

Listing 4.5: Definition of the `GroceryService`’s replicas.

Depending on the type of object that is assigned to the replica, the replica will either be available or consistent. Hence, CScript distinguishes between available and consistent data types. Objects that are of an available type result in available replicas. Whereas objects that are not of an available type result in consistent replicas.

As we will see later in this chapter, the `GroceryList` type is an available data type. Hence, Line 2 results in an available replica: `groceryList`. On

the other hand, `Inventory` is not an available data type, therefore `inventory` is a consistent replica (Line 3).

4.5.2 Consistent Replicas

Consistent replicas guarantee strong consistency, even in the face of concurrent operations. Interactions with consistent replicas may however require network communication. For this reason, property accesses and method invocations are asynchronous, as previously observed in the implementation of the `buy` method (Lines 4 to 16 in Listing 4.2).

```
1 class Inventory {
2   constructor(stock = []) {
3     this.stock = new Map(stock);
4   }
5
6   approve(itemName, stockQuantity, buyingQuantity) {
7     if (buyingQuantity <= 0)
8       return false;
9
10    const trueStock = this.stock.getOrElse(itemName, 0);
11    if (trueStock === stockQuantity) {
12      this.stock.set(itemName, trueStock + buyingQuantity);
13      return true;
14    }
15    else {
16      return false;
17    }
18  }
19 }
20
21 Map.prototype.getOrElse = function(key, notSetValue) {
22   return this.has(key) ? this.get(key) : notSetValue;
23 };
```

Listing 4.6: Implementation of the grocery inventory.

Listing 4.6 shows the implementation of the `Inventory` class. This class maintains a dictionary that maps items to the number of purchased pieces (Line 3). Since the grocery list application does not model consumptions, the number of purchased pieces equals the amount the user has in stock. Additionally, the constructor accepts an optional parameter to initialize the inventory (Line 2). This parameter must be an associative array containing `[itemName, amount]` bindings.

Inventories also define the `approve` method which validates buy requests (Lines 6 to 18). This method takes three arguments: the item's name, the stock quantity the user is aware of and the quantity the user wants to buy. First, the inventory ensures that the user wants to purchase at least one piece (Lines 7 and 8). Then on Line 10, the inventory fetches the amount that is in stock. Finally, the inventory ensures that the user's view on the stock is

consistent with the inventory’s stock. To this end, the inventory checks that the user’s `stockQuantity` argument corresponds to the `trueStock` (Line 11). If the user’s view is consistent, the inventory accepts the buy request and updates its stock (Lines 12 and 13). Otherwise, the inventory rejects the buy request (Lines 15 to 17).

To conclude, we implemented the grocery list’s inventory as a regular ES6 class. Hence, `Inventory` is not an available data type. Therefore, the `GroceryService`’s inventory is a consistent replica (Line 3 in Listing 4.5).

4.5.3 Available Replicas

Besides consistent replicas, CScript also offers native support for available replicas. Available replicas guarantee local availability and strong eventual consistency (see Section 2.3.3).

We start with an overview of some built-in available data types. Afterwards, we explain how to define new available data types. As an example, we discuss the implementation of the `GroceryList` class from our motivating example. Finally, we describe how applications react to updates.

Built-in Available Data Types

CScript contains a number of built-in available data types which can be used to create available replicas. Table 4.1 gives an overview of these data types. The interface of these data types is depicted in CScript’s class diagram (Figure 4.3). Notice that the `ORSet` and `LWWSet` data types are well-known in the literature. For a detailed explanation we refer the reader to (Shapiro et al., 2011a).

Data Type	Description
GMap	Grow-only dictionary. Cannot update or remove bindings.
ORSet	Observed-removed set data structure.
LWWSet	Last-write-wins-element set data structure.

Table 4.1: Overview of CScript’s built-in available data types.

Defining New Available Data Types

New available data types are defined by providing a concrete implementation for either the abstract `CvRDT` or `SECRO` class (see Figure 4.3). In addition,

the implemented data type must be self-contained (i.e. an isolate) and must also be registered at the factory of available data types, called `Factory`.

In the first approach, we define a new state-based CRDT (see Section 2.5) by extending the abstract `CvRDT` class. To this end, we must implement four concrete methods which are outlined below.

`merge(CvRDT): void` Instance method which merges the replica with the received `CvRDT` replica. Since the replica's state forms a join semilattice, merge updates its state to the least upper bound of both states.

`equals(CvRDT): boolean` Instance method which returns a boolean indicating whether the received `CvRDT` replica equals this replica or not. Two `CvRDT` replicas are equal if their states are equal.

`toJson(): Any` Instance method which transforms the replica into a native JavaScript representation which is serialized and disseminated over the network.

`static fromjson(Any): CvRDT` Class method which parses a replica's native representation and returns a replica instance.

In the second approach we define custom available data types using `SECRO`s. To this end, we define a concrete class which extends the abstract `SECRO` class. Recall from Section 3.1 that `SECRO`s consist of query methods and mutators. Mutators consist of the actual update operation, a precondition and a postcondition. In `CScript` query methods are decorated with `@accessor`. Pre and postconditions are defined similarly to instance methods but are prefixed with the `pre` and `post` keywords respectively.

```

1  class GroceryList extends SECRO {
2      constructor(items = []) {
3          super();
4          this.items = new Map();
5          items.forEach(this.add.bind(this));
6      }
7
8      @accessor
9      get() { /* ... */ }
10
11     add(item) { /* ... */ }
12     post add(state, originalState, args, res) { /* ... */ }
13
14     bought(itemName, quantity) { /* ... */ }
15     pre bought(state, itemName, quantity) { /* ... */ }
16
17     delete(itemName) { /* ... */ }
18
19     toJson() { /* ... */ }
20     static fromjson(items) { /* ... */ }
21 }

```

```

22
23 Factory.registerAvailableType(GroceryList);

```

Listing 4.7: Structure of the available GroceryList data type.

Listing 4.7 gives an overview of the `GroceryList`'s structure. This grocery list maintains a dictionary of items and their corresponding description (Line 4). In addition, the constructor accepts an optional argument to initialize the grocery list with some items (Lines 2 and 5).

The code uses the `@accessor` decorator to mark `get` as a side effect free query method (Lines 8 and 9). Line 12 uses the `post` keyword to define a postcondition for the `add` method. Similarly, Line 15 defines a precondition for the `bought` method.

To complete the registration of this new available data type, Line 23 registers the `GroceryList` class at the factory of available data types. From this point on, CScript treats the `GroceryList` as an available data type.

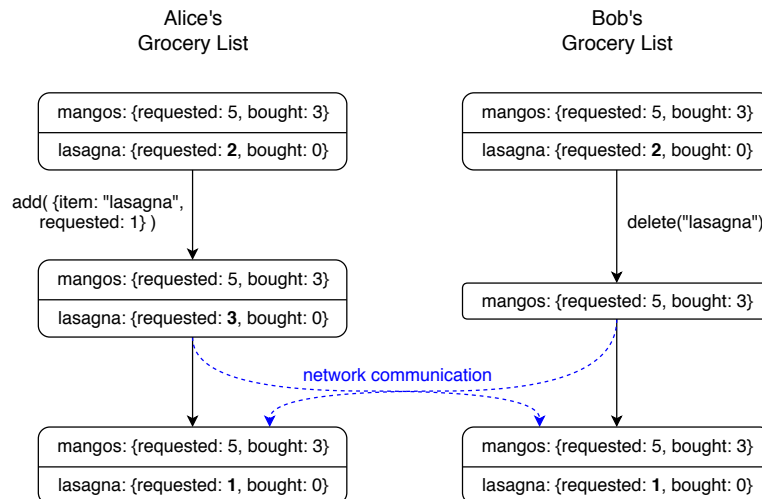


Figure 4.4: Concurrently incrementing and deleting grocery items.

Before we dive into the actual implementation of the grocery list's query and update methods, Figure 4.4 illustrates the application's expected behaviour. Initially, both Alice and Bob start with the same grocery list. Alice then requests one more lasagna, while concurrently Bob deletes the lasagnas from his grocery list. Since Bob was unaware of Alice's addition, the resulting state must contain Alice's lasagna. To achieve this behaviour, the grocery list associates a postcondition to the `add` method.

```

1 add(item) {
2   const description = this.items.getOrElse(item.name, {requested: 0,
   bought: 0});

```

```

3     description.requested += item.requested;
4     this.items.set(item.name, description);
5 }
6
7 post add(originalState, state, args, res) {
8     const [item] = args,
9         addedQuantity = item.requested,
10        resultingQuantity = state.items.getOrElse(item.name, 0).requested;
11    return resultingQuantity >= addedQuantity;
12 }

```

Listing 4.8: Adding items to a grocery list.

Listing 4.8 shows the grocery list’s `add` method and its associated postcondition. `add` starts by fetching the item’s description on Line 2. If the item does not yet exist, it creates a new item description. Afterwards, `add` increments the description’s requested quantity with the quantity of the added item (Line 3). Finally, Line 4 adds the item to the grocery list.

In order to achieve the behaviour illustrated in Figure 4.4, `add`’s postcondition must ensure that the requested quantity is added to the grocery list. In other words, Alice’s postcondition states that the resulting grocery list must contain at least one requested lasagna.

Line 7 defines `add`’s associated postcondition. The postcondition takes four arguments: the replica’s original state, the state resulting from `add`, an array containing the arguments that were received by `add` and `add`’s return value (see Section 3.1). First, the postcondition extracts the quantity from the added item (Line 9). Afterwards, it uses the replica’s resulting state to fetch the requested quantity after the operation’s execution (Line 10). Finally, Line 11 ensures that the resulting state contains at least the quantity that was added.

```

1 delete(itemName) {
2     this.items.delete(itemName);
3 }
4
5 bought(itemName, quantity) {
6     const quantities = this.items.get(itemName);
7     quantities.bought += quantity;
8 }
9
10 pre bought(state, itemName, quantity) {
11     return this.items.has(itemName);
12 }

```

Listing 4.9: Implementation of the grocery list’s `bought` and `delete` methods.

Listing 4.9 defines the `delete` and `bought` methods. Grocery items are deleted by removing the corresponding entry from the `items` dictionary (Line 2). Remember from Section 4.2.3 that `bought` informs the grocery list that a certain quantity of an item was bought. Hence, the method fetches the item’s description (Line 6) and updates the number of bought pieces

(Line 7). However, concurrent deletions may delete the item before calling `bought`. Therefore, `bought`'s precondition ensures that the item exists (Line 11).

```

1  @accessor
2  get(name) {
3    if (name) {
4      const quantities = this.items.getOrElse(name, {requested: 0, bought: 0})
5      return new GroceryItem(name, quantities.requested, quantities.bought);
6    }
7    else {
8      const items = [];
9      this.items.forEach((quantities, name) => {
10       const {requested, bought} = quantities;
11       items.push(new GroceryItem(name, requested, bought));
12     });
13     return items;
14   }
15 }
16
17 toJSON() {
18   return this.get();
19 }
20
21 static fromjson(items) {
22   return new GroceryList(items);
23 }

```

Listing 4.10: Implementation of the grocery list's `get` method and the serializable interface.

Finally, Listing 4.10 shows the implementation of the query method `get` and the `Serializable` interface (see Figure 4.3). The `get` method takes an item's name and returns that item from the grocery list (Lines 3 to 6). If no name is provided, `get` returns all items from the grocery list. To this end, `get` traverses the dictionary of grocery items (Lines 9 to 12) and accumulates all items in an array (Line 11). It then returns the array to the caller (Line 13). Additionally, the grocery list implements the compulsory `Serializable` interface. The `tojson` method uses `get` to transform the grocery list into an array representation that can be serialized³. Conversely, the `fromjson` method turns the array representation into a `GroceryList` object.

Reacting to Updates

Replicas emit two events to which applications can listen:

RemoteUpdate Replicas trigger a `RemoteUpdate` event each time the

³CScript can serialize arrays whose content is serializable. The `GroceryItem` class implements the `Serializable` interface. Therefore, the array can be serialized.

replica experiences a remote update. A remote update is an update that originates from an operation on another replica.

Update Replicas trigger an **Update** event each time the replica applies an update. Hence, this event includes both local and remote updates.

Figure 4.5 depicts a sequence diagram of our grocery application which clarifies the distinction between both events. We number the messages and refer to message i as m_i . The sequence diagram illustrates the case where Alice and Bob conceptually share a grocery list. First, Alice needs something, therefore she adds an item to the grocery list (m_1). Her grocery list replica disseminates the update to Bob's replica (m_2). Afterwards, her replica triggers the **Update** event (m_3). As a reaction to this event, the back-end informs Alice's front-end about the update (m_4), which refreshes the UI (m_5).

On Bob's side, when the replica receives the update (m_2), both the **Update** and **RemoteUpdate** events are triggered (m_7). Similarly, as a reaction to the update the back-end sends the new list to the front-end (m_8), which updates the UI accordingly (m_9).

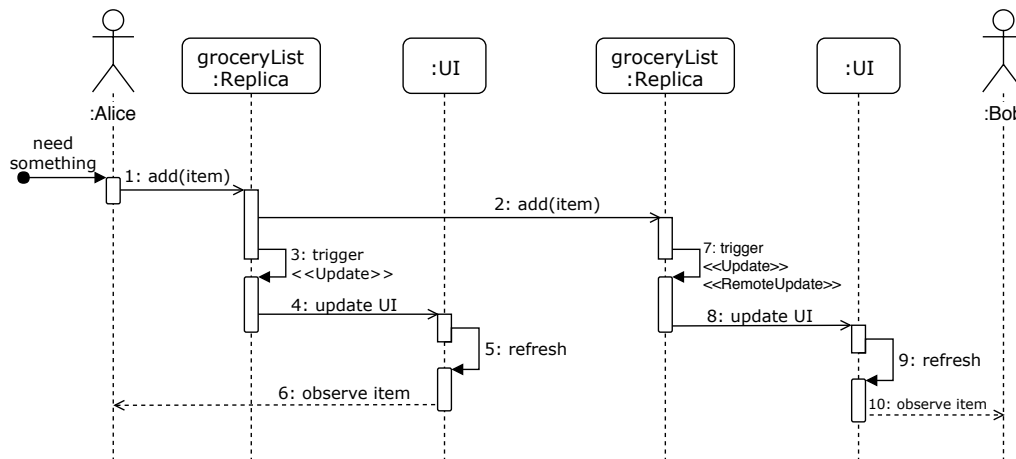


Figure 4.5: Sequence diagram illustrating updates of the grocery application.

To react to the events that are triggered by a replica, developers add listeners to those events using the replica's `onUpdate` and `onRemoteUpdate` methods.

```

1 function processService(gservice) {
2   const name      = gservice.name,
3     author       = gservice.author,
4     groceryReplica = gservice.groceryList;
5
6   groceryReplica.onUpdate(updatedList => {

```

```
7     const items = updatedList.get();  
8     updateUI(name, author, items);  
9     });  
10 }
```

Listing 4.11: Listening to update events.

Listing 4.11 shows the `processService` function. This function is called each time a new grocery service is created or discovered (Listings 4.3 and 4.4). To keep the user interface up-to-date, the function adds a listener to the `Update` event (Lines 6 to 9). Each time the replica experiences an update, the listener fetches the new grocery list (Line 7) and updates the user interface accordingly (Line 8). The `updateUI` function pushes the new grocery list to the front-end, using the list’s name and author to uniquely identify the grocery list.

4.6 Limitations

We conclude this chapter about the CScript language with a brief discussion of the language’s current limitations.

Isolates Recall that services are exchanged over the network. Therefore, services must be isolates (i.e. self-contained). However, CScript is built on top of JavaScript which does not reify the lexical environment. As a result, CScript cannot enforce services to be isolates, which places additional responsibility on the application developer. The same holds for available replicas

Inter-network communication Published services are exported on the local area network. Therefore, CScript applications need to be on the same local network in order to exchange services. If inter-network communication is required, CScript must be extended with a centralized peer and service discovery mechanism.

Op-based CRDTs CScript does not yet support operation-based CRDTs. Therefore, available data types are implemented either as state-based CRDTs or SECROs. As a result, developers are bound to simulate operation-based CRDTs. If all operations are commutative, this can be achieved using SECROs without preconditions or postconditions.

Polling Unlike available replicas, consistent replicas do not emit events when applying updates. Therefore, developers resort to traditional techniques like polling. In others words, application developers are bound to periodically query the replica in order to observe updates.

5

Implementation

This chapter describes the main design considerations behind the implementation of CScript. First, we focus on the network architecture of CScript applications, thereby explaining CScript’s peer discovery and publish-subscribe mechanisms. Afterwards, we turn our attention towards the high-level implementation of services and replicas.

5.1 The CScript Network and Communication

CScript applications are designed to run on a decentralized network where no peer lookup infrastructure may be available. Even though the network provides *zero-infrastructure*, peers need a way to discover each other. Section 5.1.1 explains how CScript achieves peer discovery.

Once peers discovered each other, they need a way to interact with one another. To this end, CScript provides a publish-subscribe mechanism which allows peers to share services. First, Section 5.1.2 describes CScript’s network architecture. Afterwards, Section 5.1.3 builds further on this knowledge to detail the implementation of CScript’s publish-subscribe scheme.

Finally, remember from Sections 2.5 and 3.2 that CRDTs and SECROs require update operations to propagate to all replicas in a causal order. Therefore, Section 5.1.4 details the implementation of CScript’s causal order broadcasting mechanism.

5.1.1 Peer Discovery

CScript's built-in peer discovery mechanism relies on UDP multicasting. Applications periodically multicast their identity over the network, thereby informing the other peers of their presence. Upon receiving a multicast message from a previously unknown peer, the application establishes a reliable bidirectional communication link with the discovered peer, i.e. one which provides exactly-once in-order message delivery.

UDP multicasting is unreliable, this implies that messages may be lost or arrive out of order. However, UDP multicasting is made reliable by continually retransmitting the messages, as explained by (Cachin, Guerraoui, & Rodrigues, 2011).

5.1.2 Network Architecture

As explained in the previous section, CScript guarantees all peers to eventually discover each other. Furthermore, applications maintain reliable bidirectional communication links with the discovered peers. As a result, applications form a *full mesh peer-to-peer overlay network*, as illustrated in Figure 5.1.

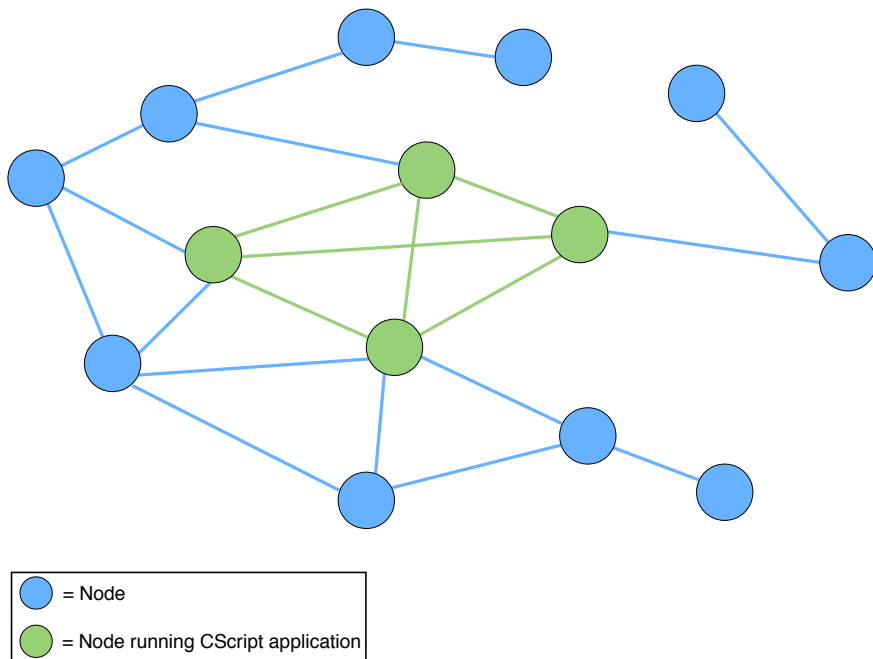


Figure 5.1: A full mesh peer-to-peer CScript overlay network.

In order to clarify the architecture shown in Figure 5.1, we outline the different aspects of a full mesh peer-to-peer overlay network:

Overlay network CScript applications are interconnected, thereby forming a logical network on top of the physical network. In Figure 5.1, circles represent the nodes of a network. Additionally, nodes running a CScript application are colored green. This collection of green nodes forms a logical overlay network on top of the underlying physical network.

Peer-to-peer system A peer-to-peer system is a collection of nodes which form a decentralized and self-organizing system (Coulouris et al., 2012). All nodes (aka peers) have roughly equal functionality, and act as both clients and servers. In other words, every node provides resources to the system (server aspect) and consumes resources from the system (client aspect). In CScript, the resources being produced and consumed are services.

Full mesh topology A full mesh topology is a network topology in which all nodes are interconnected. Hence, there is a direct communication link between every pair of nodes. The green lines in Figure 5.1 represent reliable bidirectional communication links between CScript applications.

5.1.3 Publish/Subscribe Mechanism

As explained in Section 4.4, CScript has a built-in publish-subscribe mechanism. Applications publish services using the `publish as` construct and subscribe to services using the `subscribe with` construct. Whenever a matching service is found, the application is notified. Recall that both constructs expect a type tag, which acts as a topic for the publication or subscription.

CScript’s publish-subscribe mechanism provides decoupling in time, space and synchronization. These properties result in a loose coupling between the producers and consumers of information, which is highly desirable in distributed environments (Eugster et al., 2003).

Time decoupling CScript applications can exchange services without being online at the same time. Hence, publishing a service does not require the subscribers to be online. Similarly, subscribers can discover services even though the original publisher is offline.

Space decoupling CScript applications are able to exchange services without holding references to each other. The interacting parties agree on

a type tag, which is used for service discovery. Hence, the publish-subscribe mechanism achieves decoupling in space by introducing a level of indirection.

Synchronization decoupling Publishing a service does not block the publisher's control flow. Conversely, subscribers are notified asynchronously when a matching service is discovered. Hence, CScript's publish-subscribe mechanism is decoupled in synchronization.

We now describe the implementation of CScript's publish-subscribe mechanism, and how it provides the aforementioned decoupling guarantees. To this end, we depict the various phases of publish-subscribe shown in Figure 5.2.

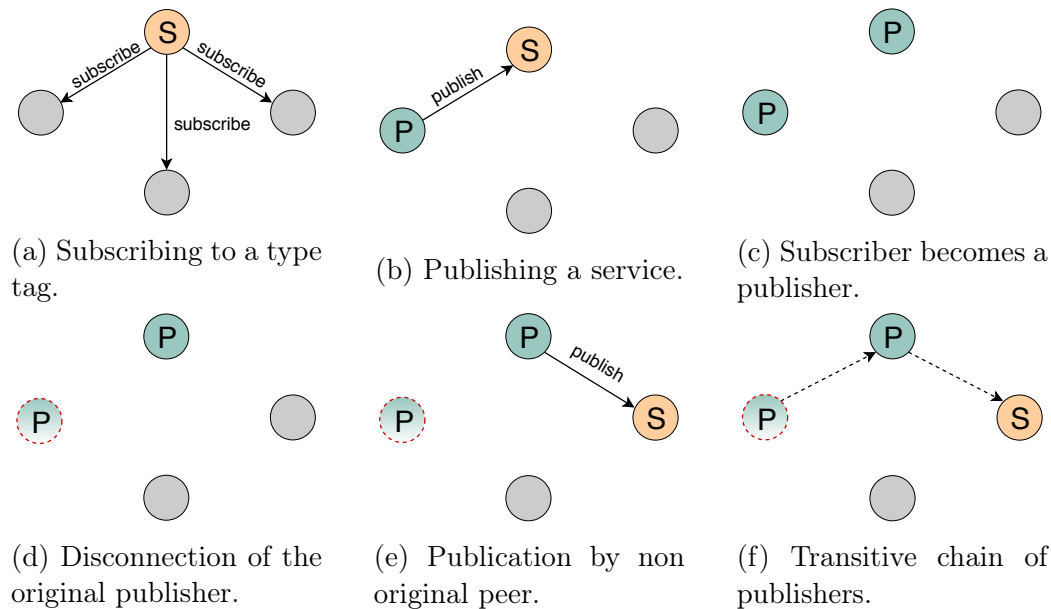


Figure 5.2: Illustration of transitivity in the publish-subscribe mechanism. Publishers and subscribers are indicated with "P" and "S" respectively, grey nodes are neither of both. Disconnected peers are indicated with a dashed red circle and gradient green color. The left node publishes a service, which eventually arrives at the right node, even though both are not online at the same time.

Peers can publish services and subscribe to services freely. Upon subscribing to services of a particular type tag, the subscriber informs all peers of his subscription (Figure 5.2a). These peers store the subscription and inform the subscriber of all previously published services which match the subscription.

Since every peer keeps track of all subscriptions, publishers are aware of all subscribers. Hence, upon publishing a service the publisher sends the service to the subscribers (Figure 5.2b).

When subscribers receive a service they automatically become publishers of that particular service (Figure 5.2c). This is possible since the subscriber has a local copy of the service. By turning subscribers into publishers we achieve transitive service discovery. For instance, assume that the original publisher disconnects (Figure 5.2d), whereafter another node subscribes (Figure 5.2e). This subscriber can get the service from the top publisher, which once also was a subscriber. Hence, this example illustrates that service discovery requires only a transitive chain of subscribers (Figure 5.2f) from the original publisher (left node) to the final subscriber (right node). As such, subscribers can discover services whose original publisher is offline. This transitive property thus guarantees decoupling in time.

Regarding decoupling in space, we previously mentioned that type tags abstract the physical location of services. As such, peers do not need to manually maintain references to each other. Additionally, CScript's publish-subscribe mechanism is decoupled in synchronization. This stems from JavaScript's concurrency model which does not allow blocking operations. Hence, publish and subscribe are non-blocking operations, and applications are notified asynchronously of a matching service, by executing the associated callback.

Theoretically, peers could run out of memory when facing too many subscriptions. This results from the fact that peers store all subscriptions. Typically, peer-to-peer applications solve this problem by distributing the data (i.e. subscriptions) over the peers of the network, in a way that ensures availability of the data. Afterwards, peers perform a distributed lookup to locate the data of interest. Well known techniques for distributing and locating data in peer-to-peer networks include flooding and distributed hash tables (Wehrle, Götz, & Rieche, 2005).

However, the limited scalability does not constitute a problem, since CScript applications run on local networks. In practice, local networks contain a limited amount of peers, making it unlikely to run out of memory. As an example, assume that every peer allocates 100MB of memory to store the subscriptions. As a bare minimum, subscriptions maintain an ip address and a type tag. Assume IPv4 addresses and type tags of maximum 20 unicode encoded characters. As IPv4 addresses take up 32 bits and a unicode encoded character may take up to 4 bytes (32 bits), we know that a subscription requires maximally $32 + 20 \times 32 = 672$ bits. Hence, the system can accommodate $\frac{8 \times 10^8 \text{ bits}}{672 \text{ bits}/\text{subscription}} = 1\,190\,476$ subscriptions.

5.1.4 Causal Order Broadcasting

Recall from Section 2.5 and Chapter 3 that operation-based CRDTs and SECROs disseminate update operations over the network. To this end, we assumed a causal order broadcasting mechanism. Hence, for any two operations o_1 and o_2 , if $o_1 <_h o_2$ then o_2 can never be delivered before o_1 . In other words, o_1 must be delivered before o_2 at all replicas. We say that $o_1 <_h o_2$ if any of the following relations apply (Cachin et al., 2011):

FIFO order Some replica r_i applies operation o_1 before applying o_2 .

Network order Some replica r_i receives operation o_1 and later applies o_2 .

Transitivity There is an operation o' such that $o_1 <_h o'$ and $o' <_h o_2$.

To ensure causal order delivery of operations, CScript relies on its full mesh network topology (see Section 5.1.2). Whenever a replica applies an update, CScript broadcasts¹ the update operation to all replicas. Upon receiving such an update operation, replicas re-broadcast the operation. This means that every replica disseminates each operation exactly once. This approach guarantees causal order delivery in all three cases listed above.

In the first case, a replica r_i applies an operation o_1 and later applies operation o_2 . Hence, the replica also broadcasts operation o_1 before o_2 . Since the underlying communication mechanism guarantees in-order message delivery, all replicas receive o_1 before o_2 .

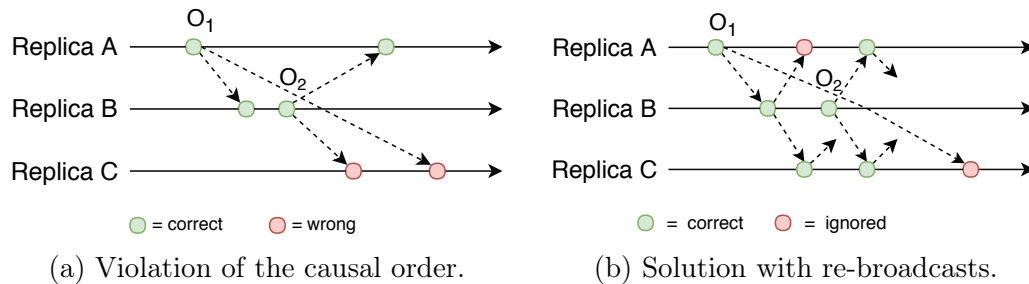


Figure 5.3: Causal order broadcasting in CScript. The time axis is drawn horizontally, with time increasing from left to right.

Figure 5.3 illustrates the second case. Replica B receives operation o_1 and later applies operation o_2 . In Figure 5.3a replica B only broadcasts operation o_2 which leads to replica C receiving operation o_2 before o_1 . This violates the

¹To broadcast the operation CScript sends the operation to each peer using the reliable communication links.

causal order. In Figure 5.3b replica B first re-broadcasts operation o_1 whereafter it broadcasts operation o_2 . Therefore, all replicas receive operation o_1 before o_2 and re-broadcast them in this order².

In the third case, operation o_1 happened before operation o' which in its turn happened before operation o_2 . Therefore, operation o_1 is broadcasted before o' and o' is broadcasted before o_2 . Hence, all three operations are delivered in causal order at the replicas.

Since the aforementioned communication scheme guarantees causal order broadcasting, we do not need to explicitly maintain dependencies (see Section 2.5). This constitutes a major advantage of this approach, since dependency sets grow unbounded over time. On the other hand, every replica broadcasts each operation once, which implies higher network traffic. However, the messages are considerably smaller.

5.2 Services And Replicas

The previous section explained the low-level implementation of CScript's peer discovery and publish-subscribe mechanisms. We now turn our attention to the high-level implementation of services and replicas. First, we explain how services and replicas are exchanged between the peers. Afterwards, we analyze the nesting of objects and replicas within other replicas and explain how CScript keeps these replicas consistent.

5.2.1 Parameter Passing Semantics

As explained in Chapter 4, services contain available and consistent replicas. When peers exchange services, the replicas must fulfill their availability or consistency guarantee. To this end, services and replicas define their own parameter passing semantics.

Exchanging Consistent Replicas

Consistent replicas must guarantee strong consistency such that all peers have a consistent view on the replica's state. In other words, once an update completes, all subsequent accesses must observe the updated value. To this end, consistent replicas are *passed by far reference* (Van Cutsem et al., 2007).

When passing a replica by far reference, the receiver acquires a proxy to the replica. Method invocations and property accesses on this proxy are relayed to the replica through asynchronous messages. As a result of this

²Replicas only re-broadcast an operation the first time they receive that operation.

design, only a single instance of the replica exists which is therefore consistent. The price to pay is that the replica forms a single point of failure. If the owner of the replica cannot be reached, the replica becomes unavailable, even for simple read accesses.

Exchanging Available Replicas

Remember from Section 4.5.3 that available replicas must guarantee availability and strong eventual consistency. Hence, the availability requirement requires peers to have a local copy of the replica which can be accessed and updated at any moment in time. However, merely copying replicas does not guarantee strong eventual consistency. Therefore, available replicas are *passed by replication*.

Pass by replication extends pass by copy with an additional consistency mechanism. When a peer discovers a service of interest, the service's available replicas are copied. Therefore, available data types must be serializable (see Section 4.5.3). CScript supports two available data types, namely state-based CRDTs and SECROs. To guarantee strong convergence updates are propagated to all replicas. When the replica is a state-based CRDT, CScript disseminates the replica's state. On the other hand, if the replica is a SECRO, CScript disseminates the update operations in a causal order.

Exchanging Services

Whenever a peer discovers a service of interest, the service is *passed by copy*. This means that methods and properties are copied, unless these properties are replicas in which case they are exchanged according to the replica's message passing semantics. This implies that services must be self-contained, i.e. isolates (see Chapter 4).

Since properties are passed by copy, updates take effect locally and do not propagate to the other copies. Hence, properties of replicated services are not kept consistent. Therefore, properties are typically used to exchange immutable data.

5.2.2 Nesting Replicas

We conclude this chapter about CScript's implementation with a brief discussion on the nesting of objects within replicas, and additionally the nesting of replicas within other replicas.

First, we consider consistent replicas, which are regular JavaScript objects that are passed by far reference (see Section 5.2.1). As a result of the pass

by far reference semantics, only a single instance of the replica exists. Hence, this replica is consistent, even though objects are arbitrarily nested. In other words, consistent replicas are strongly consistent JavaScript objects, and any nested object inherits this strong consistency property. Furthermore, available replicas cannot be nested within consistent replicas (and vice-versa), because a partition-tolerant system cannot guarantee both availability and strong consistency (see Section 2.2).

We now consider available replicas. Available replicas encapsulate their internal state. Hence, in contrast to consistent replicas, the replica's internal state is a black box which can only be accessed or updated through its public interface. In order to guarantee consistency, a replica's interface may not be circumvented. We briefly discuss how CScript enforces this.

First of all, regular JavaScript objects can be nested arbitrarily within available replicas. Nested objects belong to the replica's internal state. Therefore, programmers should not be able to manipulate nested objects directly. In other words, programmers may not hold references to nested objects. In practice, there are three ways in which programmers can obtain references to nested objects:

Passed as argument Objects can be passed as arguments to the methods of a replica. Since the replica may store the arguments internally, argument objects are considered nested within the replica. However, the programmer now holds a reference to the nested object.

Returned as value Methods of a replica may return nested objects. As such, programmers can obtain a reference to the replica's internal state.

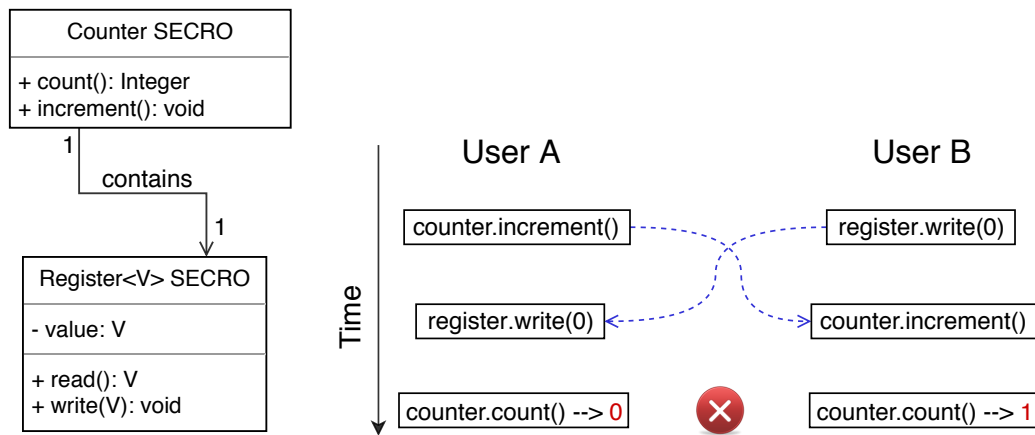
Shared object A badly designed replica may rely on an object from its lexical scope. As a result, the replica does not encapsulate the object, and the programmer can directly access the object. Therefore, we require available replicas to be isolates (see Section 4.5.3).

In order to avoid the first case, CScript deep copies the arguments that are passed to the methods of a replica. Programmers can thus continue to use the object without affecting the nested object. In fact, we make *immutable* copies, since SECROs rely on tentative executions. In other words, methods may be retried multiple times, in which case each execution must receive the same arguments as those that were originally passed by the programmer.

Similarly to the first case, CScript deep copies objects that are returned by the methods of a replica. Hence, programmers can modify returned objects without affecting the replica's internal state.

Finally, the last case should not occur since replicas must be isolates. However, CScript cannot enforce this constraint, as explained in Section 4.6.

We now turn our attention towards the nesting of available replicas. When a replica r_1 is nested within another replica r_2 , we call r_1 a nested replica and r_2 the containing replica. Nested replicas form an integral part of the containing replica’s internal state. Since replicas encapsulate their internal state, programmers cannot manipulate nested replicas directly. Instead, programmers rely on the public interface of the containing replica. As a result, nested replicas can only be mutated by their containing replica, thereby avoiding the consistency problems that arise when the containing and nested replicas can be mutated independently.



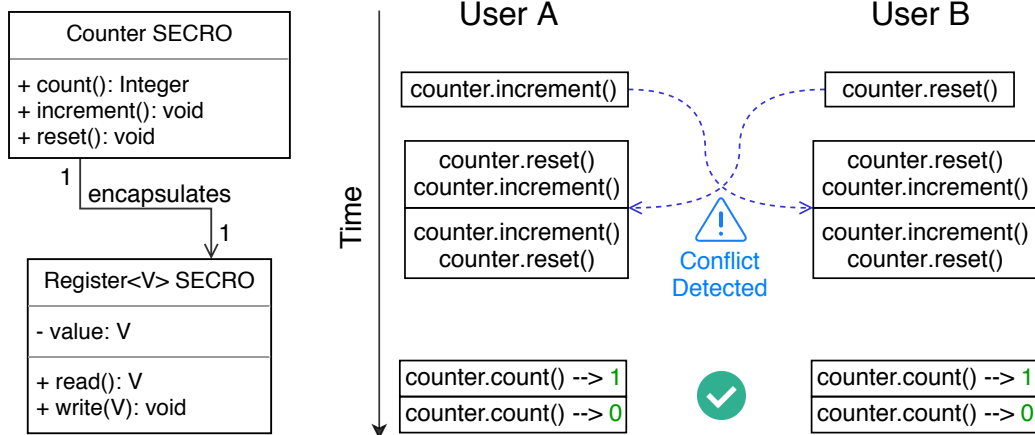
(a) Class diagram of a counter that contains a register.

(b) Replicas end up in an inconsistent state.

Figure 5.4: Example of a register replica that is nested within a counter replica. The replicas are mutated independently which leads to inconsistencies. Blue arrows indicate network communication.

Figure 5.4 shows a register replica that is nested within a counter replica. User A increments the counter while concurrently user B writes 0 to the register. Afterwards, the operations are exchanged. The problem is that the operations apply to different replicas, hence the replicas do not detect the conflict. This leads to an inconsistent state where user A observes a counter value of 0 and user B observes a counter value of 1.

Figure 5.5 depicts the same example but this time the nested replica cannot be updated directly. Instead, user B calls the counter’s reset operation which behind the scenes writes 0 to the register. When the operations are exchanged the counter replicas detect that the operations are concurrent.



(a) Class diagram of a counter that encapsulates a register.

(b) Concurrent operations are detected and re-ordered.

Figure 5.5: Avoiding consistency problems with nested replicas by allowing interactions on the containing replica only. Blue arrows indicate network communication.

Therefore, the replicas re-order the operations. This yields two possible orderings: `reset();increment()` or `increment();reset()`. As a result, when the users read the counter value they both observe the same value which is either 1 or 0.

Finally, a method of a replica may return a reference to one of its nested replicas. This leads to the problems described above. To avoid these anomalies, CScript raises an error when a method of a replica returns another replica.

6

Evaluation

This chapter evaluates our novel programming language CScript by means of a comparison between CScript’s SECRO data type and JSON CRDTs (see Section 2.6). To this end, we use a real-time collaborative text editor, which is described in Section 6.1, for both approaches. The evaluation is structured into two parts: a qualitative and a quantitative analysis. The qualitative analysis compares the implementation of the text editors, in Section 6.2. The quantitative analysis, presented in Section 6.3, consists of various experiments which quantify the memory usage, execution time and throughput of the text editors. Section 6.4 finalizes this chapter with a brief conclusion of both analyses.

6.1 Use Case: Real-time Collaborative Text Editor

Before comparing SECROs in CScript to JSON CRDTs, we implement a real-time collaborative text editor, using both approaches. Users of this application create shared text documents which can be edited simultaneously by multiple users. Collaborative text editors are the prototypical use case in the literature on CRDTs (Shapiro et al., 2011b).

Figure 6.1 depicts the class diagram of text documents. A text document

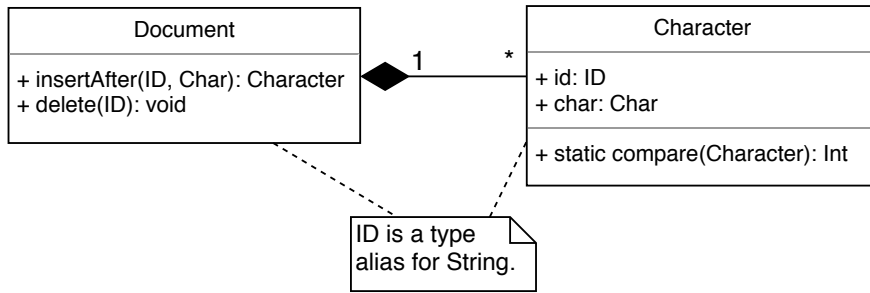


Figure 6.1: Class diagram of a text document.

is a sequence of characters, where each character is identified by a unique ID. Characters are inserted at relative positions in the text document, i.e. after another character (which is called the reference character). Therefore, the document’s `insertAfter` method expects two arguments, a character ID and the character to insert. Using the document’s `delete` method one can delete characters based on their ID. Notice that the described text editor performs single character manipulations. Hence, we insert or delete characters instead of entire words, sentences or paragraphs.

6.1.1 CScript Implementation

We now use the SECRO data type provided by CScript to implement a naive and an efficient text editor. The naive implementation stores the characters of the document in a linked list, whereas the efficient implementation organizes text documents as a balanced tree of characters.

Naive List Implementation

In the following, we present a naive text editor which organizes text documents as a linked list of characters. This means that insertions and deletions require a linear traversal of the list. For small documents this may be acceptable, however, for large documents this is not.

Listing 6.1 shows the structure of the text editor. The `Document` class extends the abstract `SECRO` class (Line 1), implements the serializable interface (Lines 17 to 24) and registers itself at the factory of available data types (Line 27).

```

1 class Document extends SECRO {
2     constructor(content = new LinkedList()) {
3         super();
4         this._content = content;
  
```

```

5     }
6
7     insertAfter(characterID, char) { /* ... */ }
8     pre insertAfter(state, characterID, char) { /* ... */ }
9     post insertAfter(originalState, state, args, newCharacter) { /* ... */ }
10
11    delete(characterID) { /* ... */ }
12    post delete(originalState, state, args, res) { /* ... */ }
13
14    @accessor
15    indexOf(id) { /* ... */ }
16
17    toJSON() {
18        return this._content; // LinkedList is a serializable type
19    }
20
21    static fromjson(content) {
22        var editor = new TextEditor(content);
23        return editor;
24    }
25 }
26
27 Factory.registerAvailableType(Document);

```

Listing 6.1: Structure of the naive SECRO text editor.

Listing 6.1 shows the `Document` SECRO. The constructor initializes the document’s contents to an empty or existing linked list (Lines 2 to 5). Lines 7 to 12 define the `insertAfter` and `delete` mutators and their associated preconditions and postconditions. Finally, the `indexOf` accessor (Lines 14 and 15) computes the index of a character based on its ID.

```

1     insertAfter(charID, char) {
2         const prev = charID === null ? 0 : this.indexOf(charID),
3               character = new Character(char, generate_id());
4         this._content.idx(prev).insertAfter(character);
5         return character;
6     }
7
8     pre insertAfter(state, charID, character) {
9         return charID === null ||
10            state.indexOf(charID) !== -1;
11    }
12
13    post insertAfter(originalState, state, args, newCharacter) {
14        const [charID, char] = args;
15        return state.indexOf(charID) < state.indexOf(newCharacter.id);
16    }

```

Listing 6.2: Implementation of the `insertAfter` mutator of the naive SECRO text editor.

Listing 6.2 shows the implementation of the `insertAfter` method and its associated precondition and postcondition. To insert a character (`char`) the method creates an internal `Character` representation and inserts it at the correct position in the linked list (Lines 3 and 4). The precondition

(Lines 8 to 11) ensures that the reference character¹ exists, whereas the postcondition (Lines 13 to 16) ensures that the inserted character occurs behind the reference character. Notice that the above implementation is conceptually identical to the replicated link list example shown in Chapter 3. Therefore, we do not go into more detail but refer the reader to Section 3.3.2 for a detailed discussion on insertions in a replicated linked list.

```
1 delete(charID) {
2     var idx = this.indexOf(charID);
3     if (idx !== -1) {
4         this._content.idx(idx)
5             .delete();
6     }
7 }
8
9 post delete(state, originalState, args, res) {
10     const [charID] = args;
11     return state.indexOf(charID) === -1;
12 }
```

Listing 6.3: Implementation of the `delete` mutator of the naive SECRO text editor.

Listing 6.3 shows the implementation of the `delete` method and its associated postcondition. To delete a character with a given ID (`charID`), the method locates the character inside the list using the `indexOf` accessor (Line 2). If the character occurs in the list, the method fetches the actual character (Line 4) and deletes it (Line 5). Additionally, the postcondition ensures that the deleted character does not occur in the list anymore (Lines 9 to 12).

Efficient Tree Implementation

We now introduce the implementation of an efficient text editor which stores text documents as a balanced tree of characters in order to support insertions and deletions in logarithmic time. Note that the text editor needs to organize the tree in a way that reflects the ordering of the characters in the document. Informally, given a node N , nodes in its left subtree must occur before N in the document, whereas nodes in the right subtree must occur after N in the document.

Although the index of a character reflects its position within the document, the text editor cannot organize the tree according to absolute character indexes, as they are not stable. In other words, absolute character indexes may change over time due to insertions and deletions. To illustrate the problem, Figure 6.2 shows a text document and its tree representation. In

¹The character after which to insert the new character.

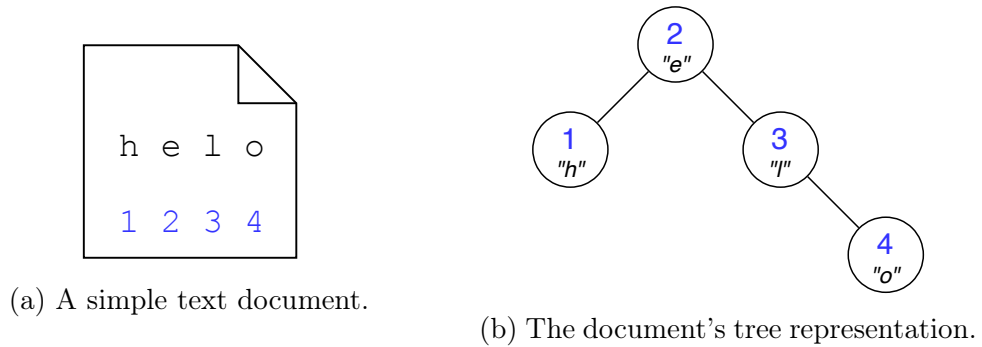


Figure 6.2: A text document and its tree representation. Numbers indicate the characters' indexes.

Figure 6.3, we insert a character in the document, which affects the index of all succeeding characters (red indexes). Hence, in order to remain correct, every affected node of the tree is updated accordingly, thereby making insertions and deletions linear operations.

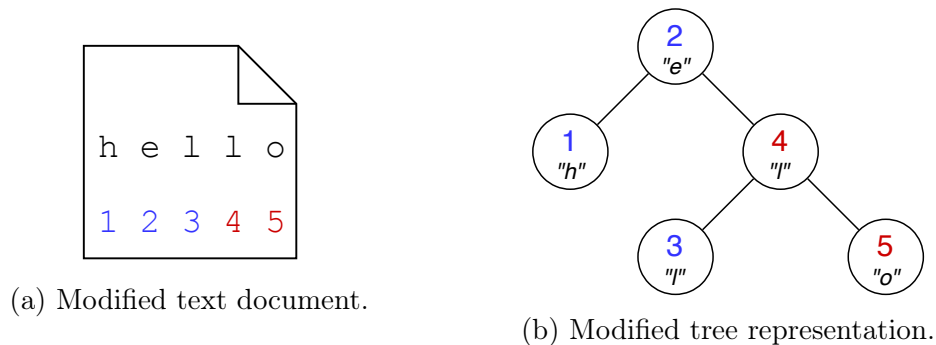


Figure 6.3: A text document and its tree representation. Red numbers indicate index changes compared to Figure 6.2.

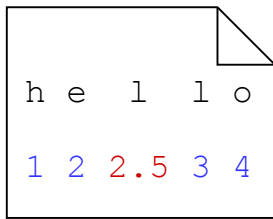
On the other hand, the unique IDs of characters are stable, however, they do not reflect the ordering of the characters. Hence, the tree cannot be organized according to character IDs either.

To solve this problem, the text editor organizes the tree according to a custom numbering scheme which **a)** reflects the position of the characters, and **b)** is stable (i.e. a character's numbering does not change over time). This scheme generates stable positions based on the positions of the previous

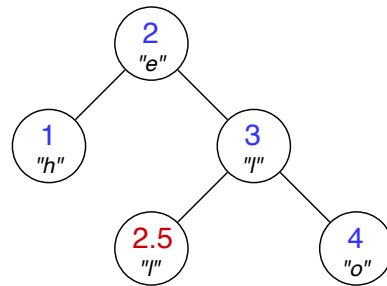
and next characters:

$$\text{new_position}(\text{prev}, \text{next}) = \begin{cases} 1, & \text{if } \neg\text{prev} \wedge \neg\text{next} \\ \frac{\text{next}}{2}, & \text{if } \neg\text{prev} \\ \text{prev} + 1, & \text{if } \neg\text{next} \\ \frac{\text{prev} + \text{next}}{2}, & \text{otherwise} \end{cases}$$

When inserting a character, its position is the average of the previous and next positions (last case). The first three cases are corner cases, which arise when the document is empty (1st case), when prepending a character to the document (2nd case) and when appending a character to the document (3rd case). Notice that this numbering scheme reflects the order of the characters since the generated position is smaller than the next position and bigger than the previous position. Furthermore, a character's position does not change over time, therefore positions can be used to uniquely identify characters.



(a) Modified text document.



(b) Modified tree representation.

Figure 6.4: A text document and its tree representation. Red number is the position of the newly added character.

Assume the same text document and positions as in Figure 6.2. If a user inserts an extra “l” before the existing “l”, this results in the text document and tree shown in Figure 6.4. Looking at the document's positions, we notice that the position of the other characters remains unchanged. Hence, we managed to organize a text document as a balanced tree of characters.

We now describe the implementation of a tree-based text editor using SECROs. The text editor relies on the AVL tree data structure provided by the Closure library (Google, 2016).

```

1 class Document extends SECRO {
2   constructor(tree = new AvlTree((c1, c2) => c1.pos - c2.pos)) {
3     this._docTree = tree;
4   }
5
6   insertAfter(pos, char) { /* ... */ }

```

```

7   pre insertAfter(state, pos, char) { /* ... */ }
8   post insertAfter(originalState, state, args, newChar) { /* ... */ }
9
10  delete(pos) { /* ... */ }
11  post delete(originalState, state, args) { /* ... */ }
12
13  @accessor
14  hasPosition(pos) { /* ... */ }
15
16  @accessor
17  determinePosition(prev) { /* ... */ }
18
19  toJSON() {
20    return this._docTree; // AVL tree is serializable
21  }
22
23  static fromjson(tree) {
24    return new Document(tree);
25  }
26 }
27
28 Factory.registerAvailableType(TextEditor);

```

Listing 6.4: Structure of the efficient text editor, which organizes its document as a balanced tree of characters.

Listing 6.4 shows the structure of the `Document` SECRO. The text editor uses the third-party AVL tree (Line 2) and turns it into a SECRO. To this end, the text editor defines the necessary preconditions and postconditions on its `insertAfter` and `delete` operations (Lines 6 to 11).

Since the implementation re-uses an existing tree data structure, the document's API forwards the insert and delete operations to the underlying AVL tree. The remainder of this section presents the implementation of the document's API, i.e. the `insertAfter`, `delete` and `hasPosition` methods.

Note that Listing 6.4 also shows two accessors: `hasPosition` and `determinePosition` (Lines 13 to 17). The former returns a boolean indicating whether or not a certain position occurs in the document. The latter uses the aforementioned numbering scheme to compute a new stable position based on the reference position.

```

1   insertAfter(pos, char) {
2     const newPos = this.determinePosition(pos),
3       newChar = new Character(char, newPos);
4     this._docTree.add(newChar);
5     return newChar;
6   }
7
8   pre insertAfter(state, pos, char) {
9     return pos === null || state.hasPosition(pos);
10  }
11
12  post insertAfter(originalState, state, args, newChar) {
13    const [pos, char] = args,
14      originalChar = {char: "dummy", pos: pos};
15    return (pos === null && state._docTree.contains(newChar)) ||

```

```
16     state._docTree.indexOf(originalChar)
17     < state._docTree.indexOf(newChar);
18 }
```

Listing 6.5: Inserting a character in a tree-based text document.

Listing 6.5 contains the implementation of the `insertAfter` method, its precondition and postcondition. The `pos` argument on Line 1 is the position of the reference character. On Line 2 the method computes a stable position for the character it is inserting. Using this position the method creates a new character on Line 3. Finally on Lines 4 and 5, the method inserts the character in the tree and returns the newly added character. Again, the precondition (Lines 8 to 10) and postcondition (Lines 12 to 18) check that the reference character exists and that the newly added character occurs at the correct position in the resulting tree.

```
1 delete(pos) {
2     return this._docTree.remove(pos);
3 }
4
5 post delete(originalState, state, args) {
6     const [pos] = args;
7     return !state.hasPosition(pos);
8 }
```

Listing 6.6: Deleting a character from a tree-based text document.

Finally, characters are deleted by removing them from the underlying tree (Lines 1 to 3 in Listing 6.6). To this end, the character's stable position is used, since it uniquely identifies the character in the tree. Afterwards, the postcondition on Lines 5 to 8 ensures that the character does not occur in the tree anymore.

6.1.2 JSON CRDT Implementation

In order to implement a collaborative text editor using JSON CRDTs, we need a JavaScript implementation of this data type. Since we could not find any, we implemented our own JSON CRDTs in JavaScript. To guarantee a fair comparison with SECROs, we used the same technology stack as is used by CScript. For a detailed description of the data type and its API, we refer the reader to the original paper by (Kleppmann & Beresford, 2017).

We now present the implementation of a collaborative text editor on top of the JSON CRDT. Recall from Section 2.6 that JSON CRDTs support two data structures: linked lists and maps. However, neither of both is suited to implement an efficient tree data structure, i.e. one which provides lookups, insertions and deletions in logarithmic time. Therefore, using JSON CRDTs programmers can only implement a naive text editor.


```

1 import {crjdt, LinkedList} from "crjdt";
2 class Document {
3   constructor() {
4     crjdt.doc = new LinkedList();
5   }
6
7   insertAfter(charID, char) {
8     const idx = charID === null ? 0 : this.indexOf(charID),
9           character = new Character(char, generate_id());
10    if (idx !== -1) {
11      crjdt.doc.idx(idx)
12        .insertAfter(character);
13    }
14    return character;
15  }
16
17  delete(charID) {
18    const idx = this.indexOf(charID);
19    if (idx !== -1)
20      crjdt.doc.idx(idx)
21        .delete();
22  }
23
24  indexOf(charID) { /* ... */ }
25 }

```

Listing 6.7: Implementation of a naive text editor using JSON CRDTs.

Listing 6.7 shows the implementation of the collaborative text editor. Upon creating a new text document, the JSON CRDT is initialized with an empty linked list of characters (Lines 3 to 5).

Similarly as before, characters are inserted after reference characters. Hence, `insertAfter` (Line 7) takes two arguments: the character to insert (`char`) and the ID of the reference character (`charID`). `insertAfter` then computes the index of the reference character (Line 8), fetches the reference character (Line 11) and inserts the new character behind it (Line 12).

Finally, characters can also be deleted from a text document. To this end, `delete` computes the character's index in the list (Line 18), fetches the character (Line 20) and finally deletes the character (Line 21).

6.2 Qualitative Analysis

We now compare the implementations of the list-based text editors, from a programming language perspective. To this end, we discuss code similarities and differences between the SECRO and JSON CRDT implementations.

Listings 6.8 and 6.9 compare the implementations of the `insertAfter` and `delete` methods. To insert a character after a reference character, both approaches rely on the API of the underlying linked list. Hence, their implementations are almost identical. Notice that the JSON CRDT version

```

1 insertAfter(charID, char) {
2     const prev = charID === null ?
3         0 : this.indexOf(charID),
4         newChar = new Character(char, generate_id());
5         this._content.idx(prev).insertAfter(newChar);
6         return newChar;
7     }
8
9     pre insertAfter(state, charID, char) {
10        return charID === null ||
11            state.indexof(charID) !== -1;
12    }
13
14    post insertAfter(oState, state, args, newChar) {
15        /*...*/
16    }
17
18    delete(charID) {
19        var idx = this.indexOf(charID);
20        if (idx !== -1)
21            this._content.idx(idx).delete();
22    }
23
24    post delete(originalState, state, args, res) {
25        /*...*/
26    }

```

Listing 6.8: Character insertions and deletions in the list-based SECRO text editor.

```

1 insertAfter(charID, char) {
2     const idx = charID === null ?
3         0 : this.indexOf(charID),
4         newChar = new Character(char, generate_id());
5         if (idx !== -1) {
6             cjrdt.doc.idx(idx).insertAfter(newChar);
7             return newChar;
8         }
9     }
10
11
12
13
14
15
16
17
18    delete(charID) {
19        const idx = this.indexOf(charID);
20        if (idx !== -1)
21            cjrdt.doc.idx(idx).delete();
22    }
23
24
25
26

```

Listing 6.9: Character insertions and deletions in the list-based JSON CRDT text editor.

performs an extra check on Line 5 to ensure that the reference character exists. In the SECRO version this check is performed by the precondition (Line 11). The main difference between both approaches is that the SECRO version requires a precondition and a postcondition (Lines 9 to 14).

Similarly, both versions rely on the API of the underlying linked list to delete characters from the text document (Line 21 in Listings 6.8 and 6.9). Hence, their implementations are almost identical. However, the SECRO version requires an additional postcondition (Line 24).

To conclude, the implementation of the list-based text editor is very similar in both approaches. The main difference lies in the fact that SECROs require the programmer to define preconditions and postconditions. In terms of lines of code, the SECRO version tends to be slightly longer. However, programmers can customize concurrent behaviour and use different data types like the tree described in Section 6.1.1, which is not possible using JSON CRDTs (see conclusion of Section 2.6).

We emphasize the fact that JSON CRDTs are not general-purpose enough to implement a tree-based text editor. In essence, the constructs provided by JSON CRDTs (lists and maps) are not sufficient to implement any kind of data type on top of them. On the contrary, SECROs can re-use any JavaScript data type. This is showcased by the SECRO implementation of the tree-based text editor (see Section 6.1.1), which uses a third-party AVL tree and turns it into a replicated data type.

6.3 Quantitative Analysis

In the second part of this chapter, we evaluate our SECRO data type by means of a comparison to JSON CRDTs. To this end, we perform a number of experiments which quantify memory usage, execution time and throughput of the previously implemented text editors. Keep in mind that SECROs were designed to ease the development of custom strong eventually consistent (SEC) data types. Hence, our goal is not to outperform JSON CRDTs, but rather to evaluate the practical feasibility of SECROs.

6.3.1 Experimental Set-up

All experiments were performed on the “Isabelle” cluster of the Software Languages Lab at the Vrije Universiteit Brussel. The specifications of this cluster are shown in Table 6.1. Depending on the nature of the experiment, it was either run on a single worker node or on all ten nodes. We specify this for each benchmark.

Root Node	
Model	HP ProLiant DL120 Gen9
CPU	Intel Xeon E5-1620 v4 @ 3.50 GHz
RAM	32 GB
SSD	200 GB
10 x Worker Node	
Model	HP ProLiant DL20 Gen9
CPU	Intel Xeon E3-1240 v5 @ 3.50 GHz
L1 Cache	256 KiB
L2 Cache	1 MiB
L3 Cache	8 MiB
RAM	32 GB
SSD	200 GB
Network	
Nodes Interconnection	10 Gbit twinax

Table 6.1: Specifications of the Isabelle cluster.

6.3.2 Methodology

To get statistically sound results we repeated each benchmark at least 30 times, yielding a minimum of 30 samples per measurement. Each benchmark started with a couple of warmup rounds, to annihilate the effects of program initialization. Furthermore, we disabled NodeJS' V8 optimizing compiler to avoid just-in-time compiler optimizations which may considerably affect the measured execution times.

Regarding the statistical analysis of our measurements, we discard samples that are affected by garbage collection, if needed for the benchmark at hand (e.g. the execution time benchmarks). Then, for each measurement including at least 30 samples, we compute the average value and the corresponding 95% confidence interval.

6.3.3 Memory Benchmarks

To compare the memory usage of the SECRO and JSON CRDT text editors, we performed an experiment in which 1000 operations were executed on each text editor. We continuously alternated between 100 character insertions followed by deletions of those 100 characters. We forced garbage collection after each operation, and measured the heap usage. The resulting measurements are shown in Figure 6.5. Green and red columns indicate character insertions and deletions respectively. Notice that forcing garbage collection

is needed to get the real-time memory usage. Otherwise, the memory usage keeps growing until garbage collection gets triggered.

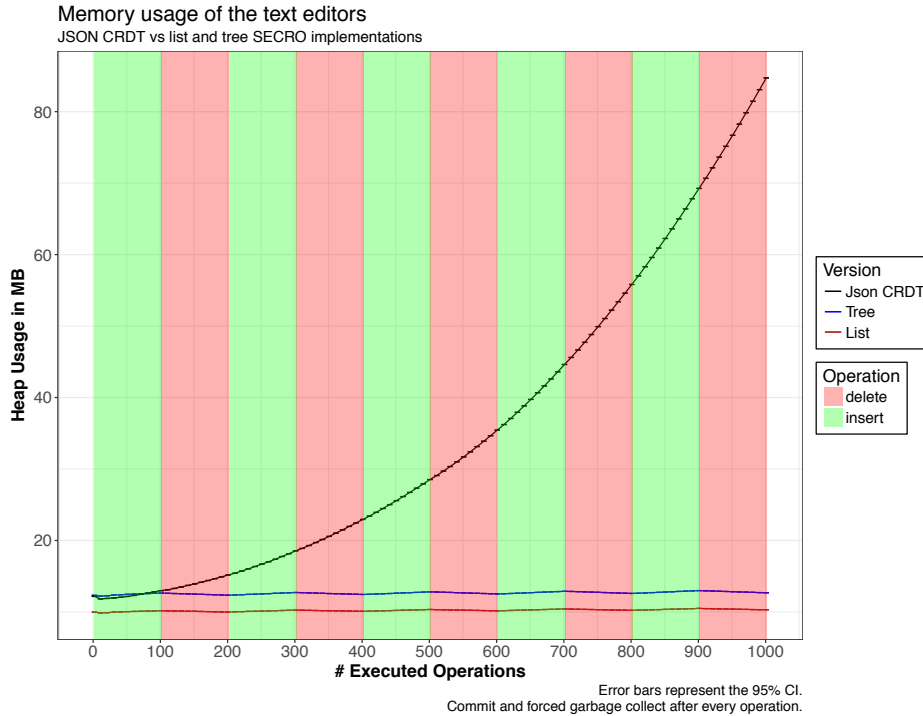


Figure 6.5: Comparison between the memory usage of the SECRO and JSON CRDT text editors. Error bars represent the 95% confidence interval for the average taken from 30 samples. This experiment was performed on a single worker node of the cluster.

Figure 6.5 confirms our expectation that the SECRO text editors are more memory efficient than the CRDT text editor. The memory usage of the CRDT text editor grows unbounded. This results from the fact the characters are not deleted. Instead, tombstones are used to mark characters as deleted. Conversely, SECROs support true deletions by re-organizing concurrent operations in a non-conflicting order. Hence, all 100 inserted characters are deleted by the following 100 deletions. This results in lower memory usage.

Figure 6.6 compares the memory usage of the list and tree text editors. We conclude that the tree implementation consumes more memory than the list implementation. The reason is that nodes of a tree maintain pointers to their children, whereas nodes of a singly linked list maintain only a single pointer to the next node. Secondly, we observe a staircase pattern. This pat-

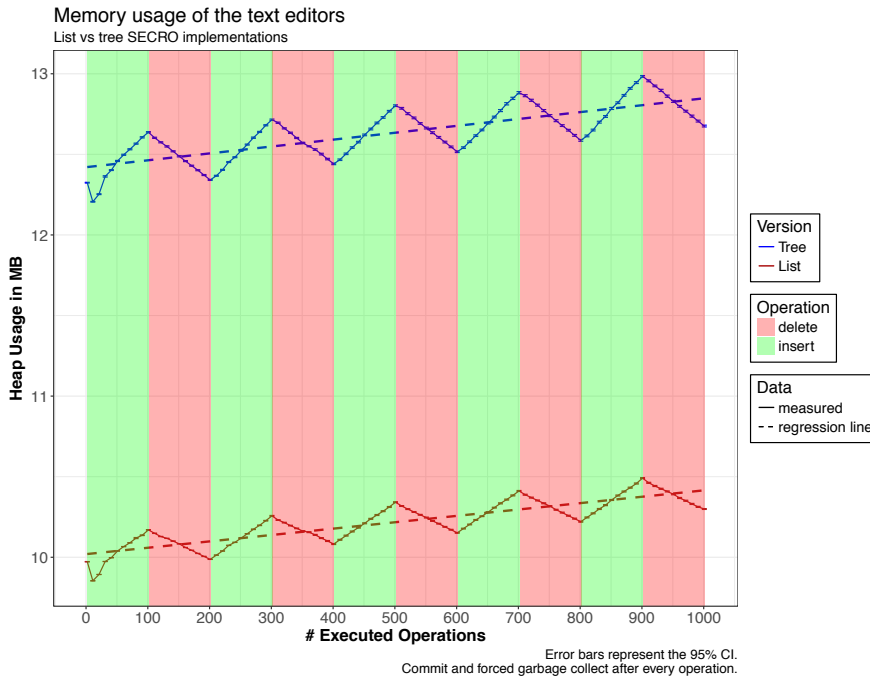


Figure 6.6: Comparison between the list and tree implementations of the SECRO text editor. Error bars represent the 95% confidence interval for the average taken from 30 samples. This experiment was performed on a single worker node of the cluster.

tern indicates that memory usage grows when characters are inserted (green columns) and shrinks when characters are deleted (red columns). Finally, memory usage increases linearly with the number of executed operations, even though we delete the inserted characters and commit the replica after each operation. Hence, SECROs entail a small memory overhead for each executed operation. This linear increase is shown by the dashed regression lines.

6.3.4 Execution Time Benchmarks

We now present a number of experiments which quantify the execution time of operations. First, we analyze the performance overhead of SECROs using an artificial example. Afterwards, we compare the performance of the SECRO and JSON CRDT text editors. Note that all experiments were performed on a single worker node of the cluster.

Overhead of SECROs

To quantify the performance overhead of SECROs we measure the execution times of 500 constant time operations, for different commit intervals. Each operation computes 10 000 tangents and has no associated precondition or postcondition. Hence, the resulting measurements reflect the best-case performance of SECROs.

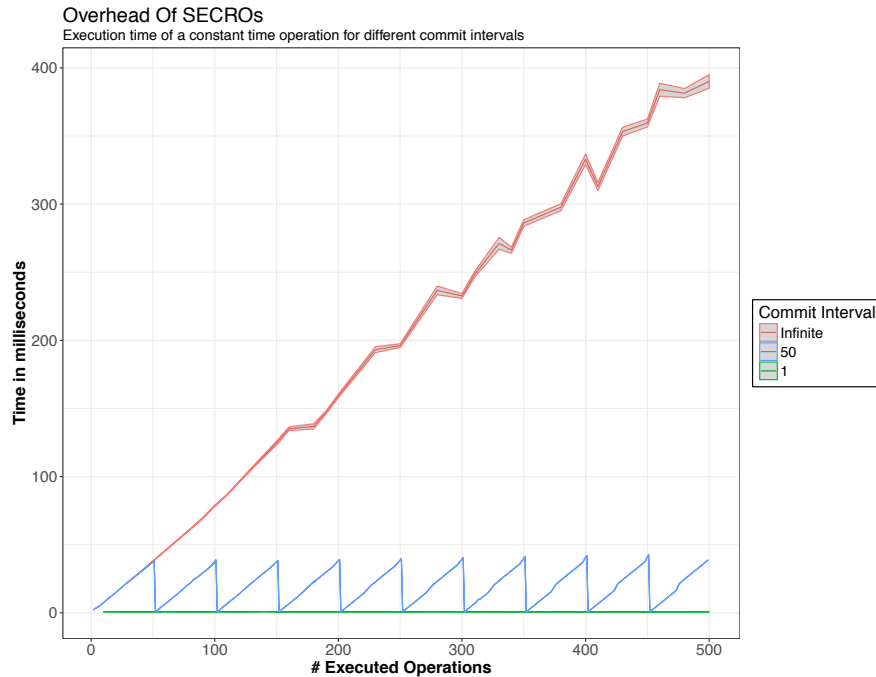


Figure 6.7: Execution time of a constant time operation in function of the number of experienced operations, for different commit intervals. Error bands represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded.

Figure 6.7 depicts the execution time of the aforementioned constant time operation. If we do not commit the replica (red curve), the operation’s execution time increases linearly with the number of operations. Hence, SECROs induce a linear overhead. This means that operations which can be performed in constant time are performed in linear time by our SECRO data type. This results from the fact that the replica’s operation history grows with every operation. Furthermore, each operation requires the replica to reorganize the history. To this end, the replica generates permutations of the history until a valid ordering of the operations is found (see Listing 3.11).

Since we defined no preconditions or postconditions, every order is valid. The replica thus generates exactly one permutation and validates it. To validate the ordering, the replica executes each operation (see Listing 3.12). Therefore, the operation's execution time is linear to the size of the operation history. In this case, the operation history contains all operations.

Remember from Section 3.5 that commit implies a trade-off between concurrency and performance. Small commit intervals lead to better performance but low concurrency, whereas large commit intervals support more concurrent operations at the cost of performance. This benchmark illustrates the performance aspect of the trade-off.

For a commit interval of 50 (blue curve), we observe a sawtooth pattern. The operation's execution time increases until the replica is committed, whereafter it falls back to its initial execution time. This is because *commit* clears the operation history.

When choosing a commit interval of 1 (green curve), the replica is committed after every operation. Hence, the history contains a single operation and does not need to be reorganized. This results in a constant execution time, independent of the number of experienced operations.

In conclusion, SECROs induce an important overhead on the execution time of operations if replicas are not committed periodically. The rate at which replicas are to be committed depends on the application at hand.

Appending Characters to a Text Document

We now analyze the performance of character insertions using the list implementation of the SECRO text editor. Figure 6.8 shows the time it takes to append a character to the text document in function of the document's length. Notice that a document length of 200 implies that 199 insertion operations preceded the depicted operation. If we do not commit the replica (red curve), append exhibits a quadratic execution time. This is because the SECRO induces a linear overhead (see previous section) and append is a linear operation. Hence, append's execution time becomes quadratic.

For a commit interval of 100 (blue curve) we again observe a sawtooth pattern. In contrast to Figure 6.7 the peaks increase linearly with the size of the document, since append is a linear operation.

If we choose a commit interval of 1 (green curve) we get a linear execution time. This results from the fact that we do not need to re-organize the replica's history. Hence, we execute only a single append operation.

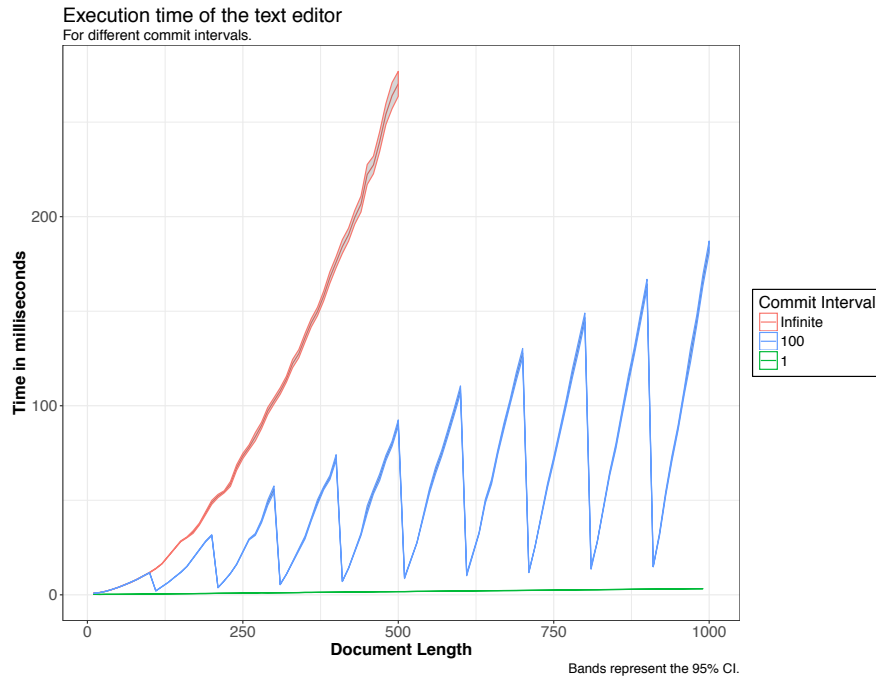


Figure 6.8: Time to append a character to the text document in function of the document’s length. Error bands represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded. This experiment was performed on the list implementation of the SECRO text editor.

SECRO vs JSON CRDT Text Editor

We now compare the performance of the naive SECRO and JSON CRDT text editors. To this end, we appended 1000 characters to a text document, and measured the time of each append operation. For the SECRO version we committed the replica after each operation.

From Figure 6.9 we conclude that character insertions exhibit a linear time complexity in both versions. Notice however that the JSON CRDT text editor is more performant, since the execution time grows less fast (i.e. has a smaller slope).

List vs Tree Text Editor

Remember from Section 6.1.1 that we implemented two text editors in the SECRO approach. The first version is a naive text editor which organizes the document as a linked list of characters. The second is an efficient text

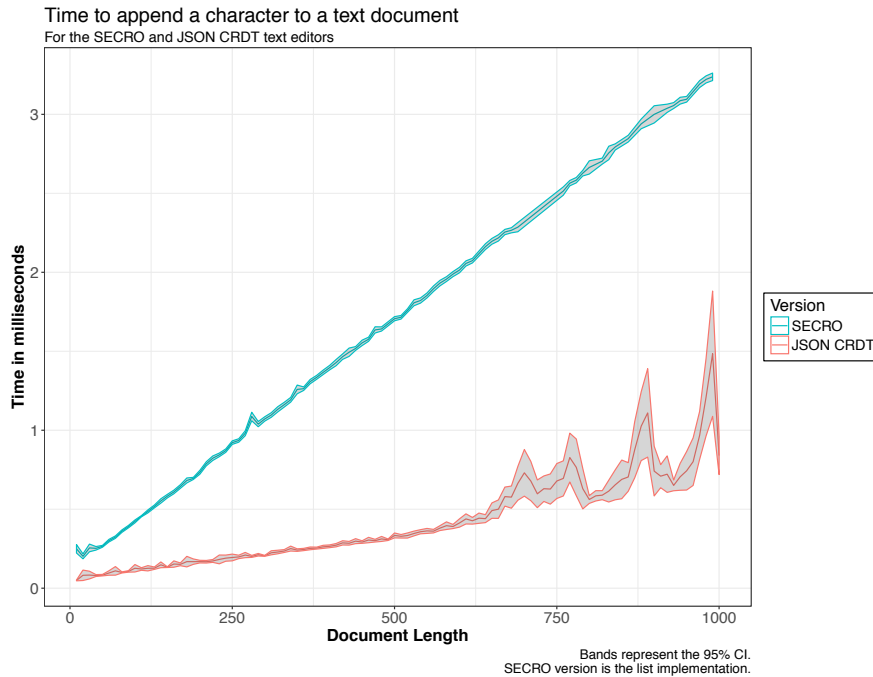


Figure 6.9: Time to append a character to a document, for the naive SECRO and JSON CRDT text editors. The SECRO version committed the replica after each append operation. Error bands represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded.

editor which organizes the document as a balanced tree of characters in order to provide logarithmic time lookups, insertions and deletions.

Figure 6.10 depicts the time it takes to append a character to the document. Contrary to our expectations, the list implementation is faster than the tree implementation. To determine the cause of this counterintuitive observation we measured the different parts that make up the total execution time:

Execution time of operations Total time spent on executing append operations.

Execution time of preconditions Total time spent on executing preconditions.

Execution time of postconditions Total time spent on executing postconditions.

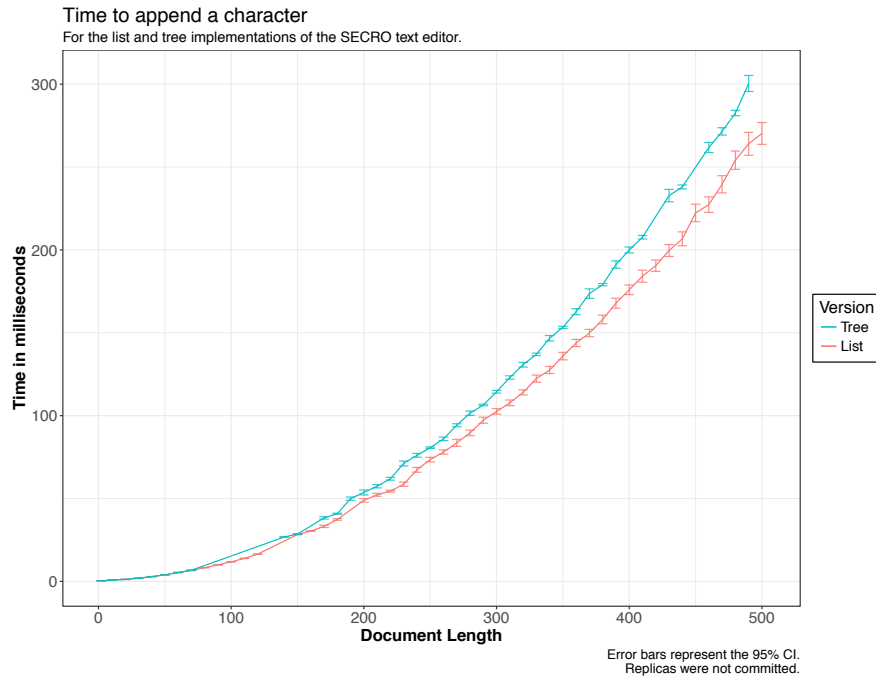


Figure 6.10: Time to append a character to a document, for the list and tree implementations of the SECRO text editor. Replicas were never committed. Error bars represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded.

Copy time As explained in Section 3.4.2, each operation requires the replica to reorganize its history and then tentatively validate this history. To validate the history (see Listing 3.12), the replica deep copies the object whereafter the operations, preconditions and postconditions are executed on the copy. Furthermore, the replica copies the object before each precondition. The total time spent on copying objects (i.e. the document) is the copy time.

Figures 6.11a and 6.11b depict the detailed execution time for the list and tree implementations respectively. One immediately observes that the total execution time is dominated by the copy time. Furthermore, the tree implementation spends considerably less time executing operations, preconditions and postconditions, than the list implementation. This results from the fact that the balanced tree provides logarithmic time insertions whereas insertions in a linked list are linear. On the other hand, the tree implementation

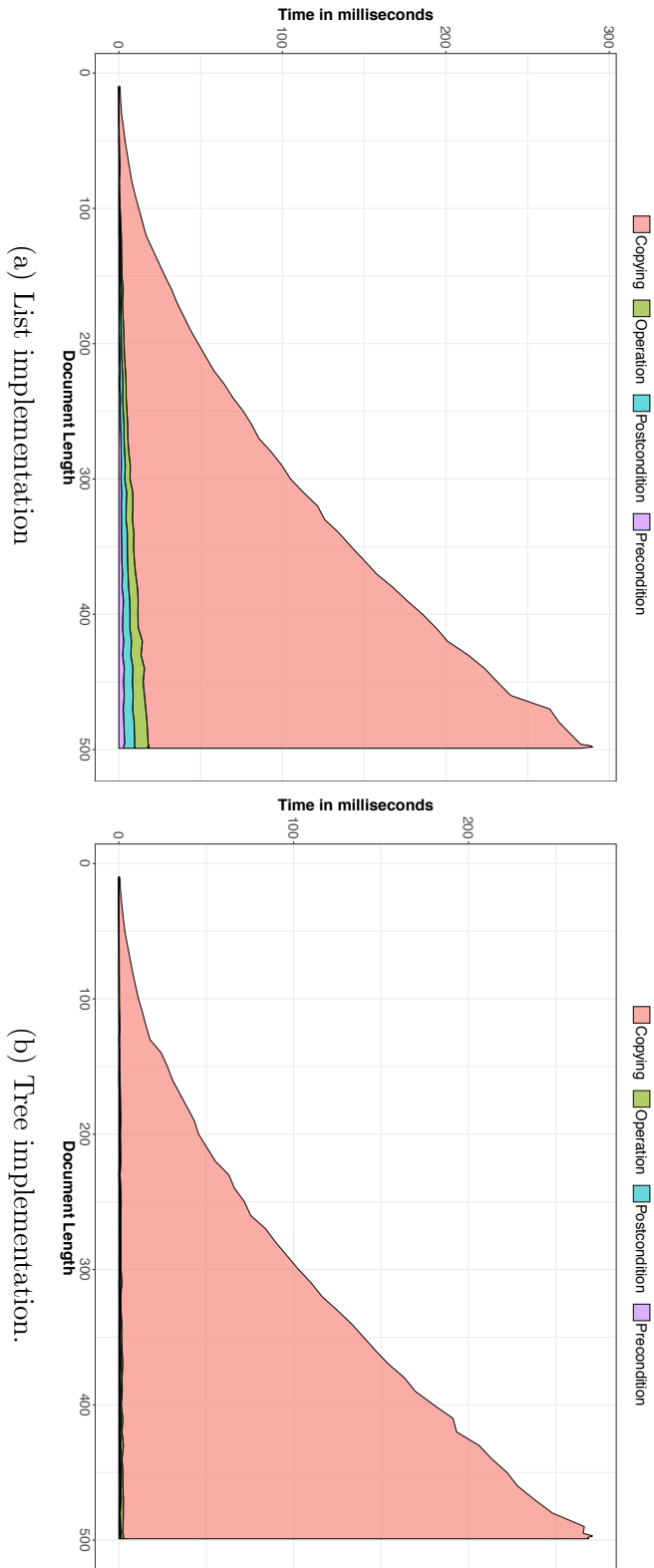


Figure 6.11: Detailed execution time for appending characters to the SECRO text editor. The replica was never committed. The plotted execution time is the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded.

spends more time on copying the document than the list implementation. The reason for this is that copying a tree entails a higher overhead than copying a linked list, since more pointers need to be copied.

A crucial insight is that each operation appends a single character to the document. Hence, for each insertion the replica first copies the entire tree of characters. This incurs a time overhead which kills the speedup gained from logarithmic time insertions.

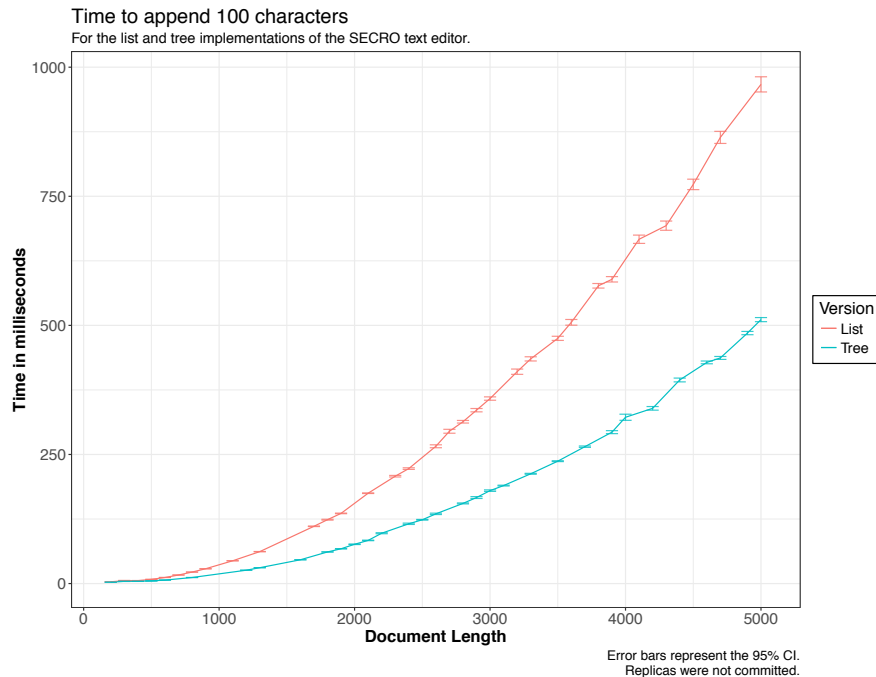


Figure 6.12: Execution time of an operation which appends 100 characters to a text document, in function of the document’s length. Replicas were never committed. Error bars represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded.

To validate this hypothesis, we re-executed the benchmark shown in Figure 6.10 but this time each operation inserts 100 characters. Figure 6.12 shows the resulting execution times. As expected, the tree implementation now outperforms the list implementation. The reason for this is that the speedup obtained from 100 logarithmic insertions exceeds the copying overhead induced by the tree. In practice, this means that single character manipulations are too fine-grained. Instead, we must switch to insertions and deletions of entire words, sentences or even paragraphs.

From the previous benchmarks we know that deep copying the document raises a considerable overhead. However, we are forced to copy the document since JavaScript objects are mutable. Hence, this problem is not inherent to SECROs, but rather a side-effect of JavaScript's mutability. To decouple this problem from SECROs, we now quantify the copy overhead induced by JavaScript.

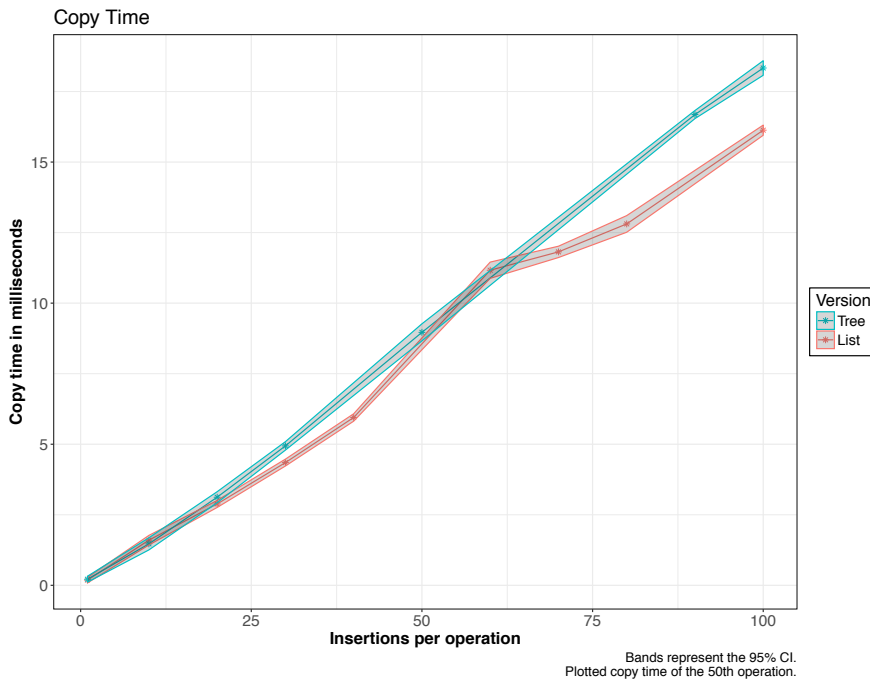


Figure 6.13: Time to copy a document in function of the number of insertions per operation. The replica was committed after each operation. For each configuration (1, 10, or more insertions per operation) we executed the operation 50 times and measured the copy time of the 50th execution. Measurements are indicated using asterisks. Error bands represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded.

Figure 6.13 plots the copy time for varying number of insertions per operation. We conclude that the copy time is linear to the number of insertions per operation. Furthermore, the copy time is slightly higher for documents which are organized as a tree of characters (as previously observed in Figures 6.11a and 6.11b). The reason for this is that nodes of a tree maintain pointers to their children, whereas nodes of a singly linked list only maintain a single pointer to the next node.

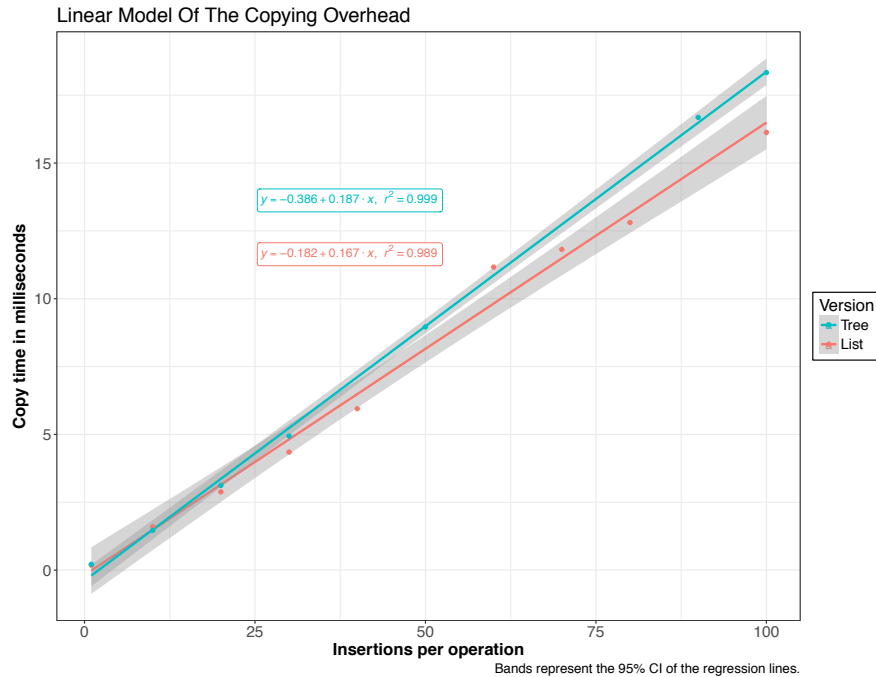


Figure 6.14: A simple linear regression model of the copy time. Error bands represent the 95% confidence interval for the regression line. Measurements are the average of at least 30 samples and are indicated by dots. Samples affected by garbage collection were discarded.

Figure 6.14 quantifies the copy overhead by fitting a linear model to our measurements. For both versions the model is a good representation since the coefficients of determination (r^2) are close to one. Based on the slopes of the linear models we conclude that the tree implementation induces an additional copy overhead of 0.02ms per insertion, compared to the list implementation.

We expect that the copy time overhead can be avoided altogether by switching to an immutable language. As such, we do not need to explicitly copy objects. Instead, operations produce new objects which extend the original object.

6.3.5 Throughput Benchmarks

The experiments presented in the previous section focused on the execution time of sequential operations on a single node. We now transition to a distributed scenario, and measure the throughput of the text editors under high computational loads.

To this end, we use all 10 worker nodes of the cluster and let them simultaneously perform a considerable amount of operations on the text editor. The total amount of operations is equally spread over the nodes of the cluster. We then measure the time to convergence, i.e. the time that is needed for all nodes to process all operations and reach a consistent state. As an example, if we have a total of 1000 operations, each node executes 100 operations. Assuming that the nodes converge after 8 seconds, we find a throughput of $\frac{1000}{8} = 125$ operations per second.

SECRO vs JSON CRDT Text Editor

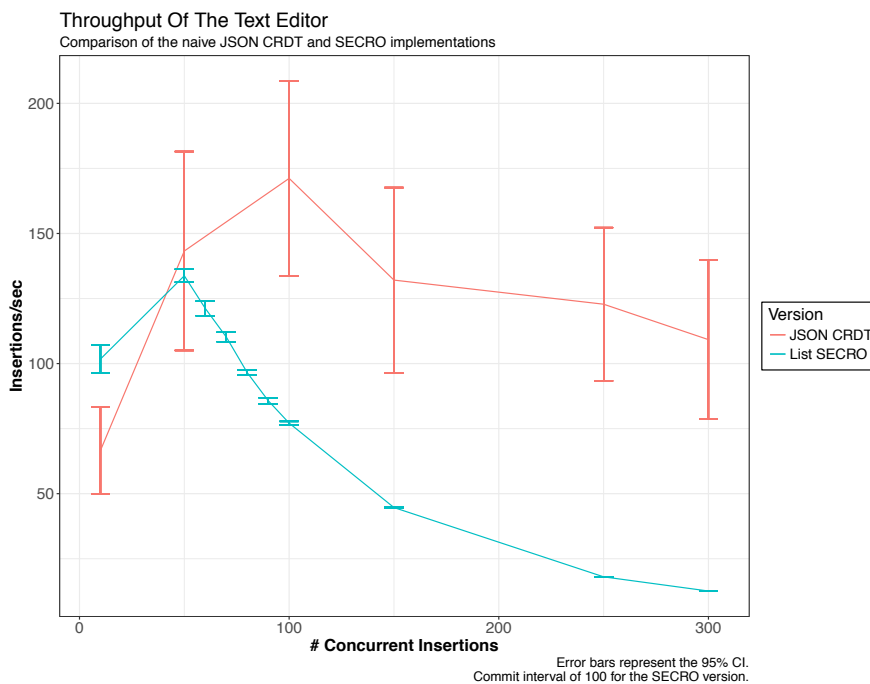


Figure 6.15: Throughput of the naive SECRO and JSON CRDT text editors, in function of the number of concurrent operations. The SECRO version committed the document replica at a commit interval of 100. Error bars represent the 95% confidence interval for the average taken from 30 samples.

Figure 6.15 depicts how the throughput of the naive text editors varies in function of the load. We notice that the SECRO text editor scales up to 50 concurrent operations, at which point it reaches its maximal throughput. Afterwards, the throughput quickly degrades. Additionally, we observe that under high loads (100 concurrent operations and more) the JSON CRDT

version achieves a higher throughput than the SECRO version. Hence, the JSON CRDT text editor scales better than the SECRO text editor.

List vs Tree Text Editor

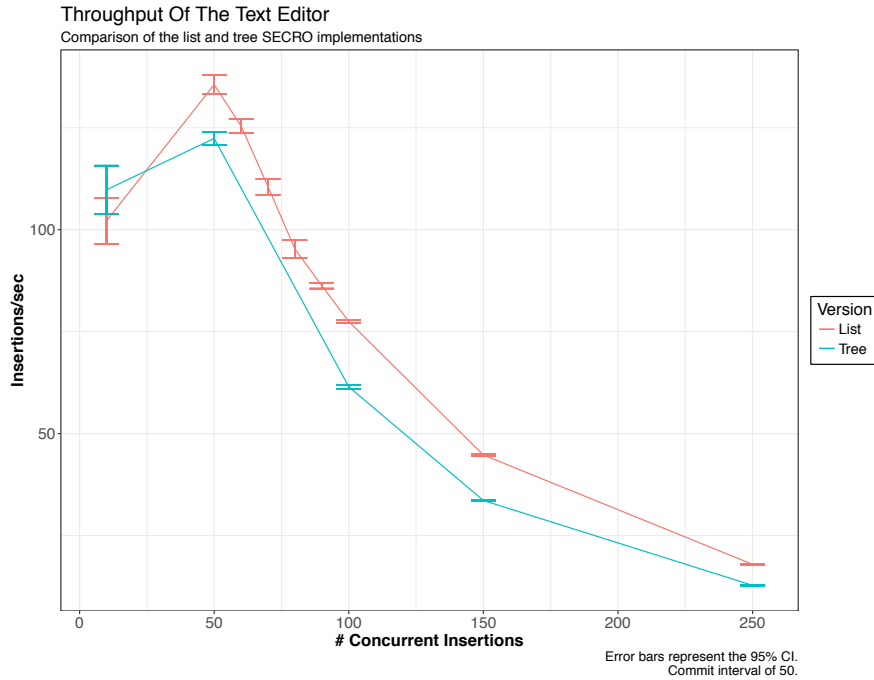


Figure 6.16: Throughput of the list and tree SECRO text editors, in function of the number of concurrent operations. Replicas were committed every 50 insertions. Error bars represent the 95% confidence interval for the average taken from 30 samples.

Figure 6.16 compares the throughput of the list and tree implementations. Both versions scale up to 50 concurrent operations, whereafter the throughput degrades. We also notice that the list implementation achieves slightly higher throughput rates than the tree implementation. The reason for this is that each operation inserts only a single character. Hence, the copy time overhead exceeds the speedup we get from organizing the document as a tree (see Section 6.3.4).

6.4 Conclusion

We evaluated CScript by comparing its novel SECRO data type with JSON CRDTs. At the basis of this comparison lies a real-time collaborative text editor, which is the prototypical example in the literature on CRDTs (Shapiro et al., 2011b). The evaluation consisted of two parts: a qualitative and a quantitative analysis.

The qualitative analysis revealed that both approaches yield similar implementations. The main difference is that SECROs require the programmer to define preconditions and postconditions. These are necessary for SECROs to provide strong eventual consistency while being a truly general-purpose replicated data type. On the other hand, JSON CRDTs are not general-purpose enough to implement any kind of applications. We demonstrated this by implementing a collaborative text editor that organizes text documents as a balanced tree of characters. SECROs require only minor changes to the list-based text editor, whereas, JSON CRDTs cannot reproduce this example at all.

In the quantitative analysis we compared both approaches in terms of memory usage, execution time and throughput. To this end, we performed a multitude of experiments using the previously implemented text editors.

The memory benchmarks revealed that the SECRO text editor consumes considerably less memory than the JSON CRDT text editor. The memory usage of the JSON CRDT version actually grows unbounded due to the use of tombstones.

Regarding the execution time of operations, the experiments showed that SECROs induce a linear overhead which is proportional to the size of the operation history. This results from the fact that replicas validate their history for each operation. We have seen that this overhead can be kept within acceptable bounds by periodically committing the replica. Commit implies a trade-off between performance and concurrency. In many applications concurrency is limited, Google Docs for instance allows a maximum of 100 simultaneous collaborators (Google, 2018). Hence, deciding on an appropriate commit strategy depends on the application and needs to be fine-tuned by the programmer.

We also compared the list and tree implementations. At first, the tree implementation did not provide the expected speedup because single character insertions are too fine-grained. However, by inserting more characters per operation a considerable speedup was achieved. With 100 insertions per operation and a document length of 5000 characters the tree implementation is roughly twice as fast as the list implementation. In practice, this means that operations of the text editor should manipulate entire words, sentences

or even paragraphs.

Finally, we measured how the text editor's throughput varies in function of the load. We found that the JSON CRDT text editor scales better than the SECRO text editor. This results from the fact that JSON CRDTs can directly apply the operations it receives, whereas SECROs need to re-order the operations which can be costly if there are many concurrent operations. JSON CRDTs thus achieve better scalability at the cost of generality. On the other hand, the scalability of SECROs needs to be fine-tuned, but one has a truly general-purpose available replicated data type.

7

Conclusion

In this dissertation, we explored programming language support for availability and consistency of distributed applications. Remember that distributed systems replicate data to improve the availability, performance and scalability of the system. However, when updating a copy it becomes different from the other copies. This means that programmers need to keep the copies consistent to some extent.

Keeping the copies strongly consistent requires agreement between the nodes that make up the system such that they update the copies consistently. This forces the programmer to implement a distributed consensus algorithm (Lamport, 1998; Ongaro & Ousterhout, 2014) which is a challenging task. Furthermore, users cannot issue updates while being offline, since the other copies cannot be informed of the update. Hence, strong consistency comes at the cost of availability.

Another possibility is to favor availability over strong consistency such that users can continue to use the system even though they are offline. However, updates may now lead to inconsistencies which need to be resolved by the application developer. Solving these conflicts is a difficult task and depends on the behaviour that is expected from the application. For this reason, the literature provides no general-purpose data type for implementing arbitrary available data structures.

Although the aforementioned problems are faced by most distributed

programmers, there is no distributed programming language that aids the programmer with the development of both consistent systems and available systems. The development of such a language raises a number of essential research questions which are central to this dissertation:

RQ. 1 Which language constructs are needed to simplify the development of both available systems and consistent systems, and how can we integrate these constructs in one distributed programming language?

RQ. 2 Is it possible to design a general-purpose data type that guarantees availability?

7.1 Our Approach

To address the aforementioned research questions we design and implement CScript, a next generation distributed programming language. CScript tackles the problems of availability and consistency at the level of objects. Objects are kept strongly consistent by serializing all updates on a single copy of the object. This object acts as the master copy. On the other hand, objects are made available by allowing updates to happen concurrently on different copies of the object. In order to keep these copies consistent, we propose the use of state validators.

State validators are language constructs for expressing invariants over the state of an object. Programmers use state validators to specify the behaviour that is expected from available objects in the face of concurrent operations. The idea is to use this information to detect and solve the conflicts that arise. As a proof of concept, CScript provides a generic object data type for implementing available data structures that behave accordingly to the declared invariants.

7.1.1 A Distributed Object-oriented Model for Replication

The goal of our research is to augment the object-oriented programming model with dedicated language support for replication. The key idea is to design one language (CScript) that offers objects that can be replicated and for which the programmer can specify whether the object should be available or strongly consistent. If the object is available it will be only eventually consistent. Conversely, strongly consistent objects are not fully available upon network partitions.

In CScript, replicated objects are called replicas. There are two types of replicas, namely available and consistent replicas. Available replicas cannot be nested within consistent replicas and vice versa. This results from the CAP theorem which states that a system cannot provide both guarantees when facing network partitions.

CScript also allows programmers to combine replicas into larger components, called services. Services offer specific functionality and can be shared between the different peers of the system. Upon sharing a service with a peer, the peer receives a copy of the service. CScript ensures that the service's replicas fulfill their availability or consistency guarantee.

Consistent Replicas

To ensure that consistent replicas remain strongly consistent, CScript maintains a single copy of the object and serializes all updates on that copy. This implies that in the face of disconnections, updates are buffered until reconnection. Developers thus perceive these objects as available, however, operations are buffered.

Available Replicas

In CScript, available replicas are replicated to all peers of the distributed system. This guarantees availability since every peer has a local copy from which he can read or write.

However, conflicts can arise if two or more peers concurrently update their local copy. Even though these updates are individually correct, their combination may violate some invariant leading to an inconsistent state (Shapiro et al., 2011b). To make these copies consistent again the conflict must be resolved.

To deal with these problems, available replicas are *passed-by-replication* in CScript. This means that the replica is passed-by-copy to the receiver and that dedicated mechanisms are used to keep the copies consistent to the extent possible. In contrast to consistent replicas, users may observe temporal inconsistencies, however, at some point in time the copies become consistent again.

The consistency guarantees provided by a system are described by its *consistency model*. Such models restrict the values that a read operation may return (Tanenbaum & Van Steen, 2007). The implementation of a consistency model is called a *consistency protocol*.

In CScript, available replicas implement the strong eventual consistency (SEC) model. This model prescribes that replicas that received the same

operations, possibly in a different order, are consistent. CScript provides two abstractions that rely on two different protocols for this model, namely conflict-free replicated data types (CRDTs) and strong eventually consistent replicated objects (SECROs).

CRDTs CRDTs are special data types that require concurrent operations to commute. Conflicts are thus avoided by design since any ordering of the operations yields the same outcome. However, imposing operations to commute is a severe restriction which does not allow programmers to implement any kind of available data structure.

SECROs A SECRO is a general-purpose data type for implementing arbitrary available data structures. Programmers use dedicated language constructs, called state validators, to specify the concurrent behaviour of SECROs through a set of invariants. Since operations do not necessarily commute, reaching a consistent state requires the replicas to execute all operations in the same order. Therefore, SECROs re-order concurrent operations based on the declared invariants. As such, concurrent operations cannot lead to unexpected behaviour.

To re-order the operations, we designed a deterministic algorithm that yields the same order at all replicas, independent of the order in which the operations were received and without having to communicate between the replicas. CScript embeds a proof-of-concept implementation of SECROs.

7.1.2 Evaluation

To evaluate CScript, we compare SECROs with the JSON CRDT (Kleppmann & Beresford, 2017), which is a CRDT implementation of a general-purpose data type. Although the JSON CRDT does not directly provide replicated objects, it allows developers to implement custom CRDTs without having to deal with conflicts. Hence, the JSON CRDT is said to generalize CRDTs to any general-purpose data structure. In particular, we compare CScript’s SECROs to JSON CRDTs by means of a collaborative text editor application. The application was proposed in the original JSON CRDT paper (Kleppmann & Beresford, 2017) and is the prototypical example in the literature on CRDTs.

Our evaluation consists of two parts: a qualitative and a quantitative analysis. The qualitative analysis compares the implementation of the SECRO and JSON CRDT text editors. We found that the main difference lies

in the fact that JSON CRDTs hardcode concurrent behaviours, whereas SECROs rely on the invariants defined by the programmer. As a result, the SECRO implementation yields slightly more lines of code but is more flexible. This means that programmers can customize the concurrent behaviour of replicated objects as to match the needs of the application, which is not possible using JSON CRDTs. Hence, SECROs are truly general-purpose but require developers to define state validators. This places the responsibility of developing correct data types on the programmer but at a much higher level of abstraction than CRDTs. Developers declare what is expected from the data type, rather than specifying how low-level conflicts must be solved or avoided.

To demonstrate the flexibility of SECROs we implemented a text editor that stores documents as a tree of characters. The SECRO implementation re-uses a third-party AVL tree and turns the tree into an available replicated data type. This showcased the flexibility of CScript to convert any JavaScript data type into a SECRO in just a few steps. On the opposite, using JSON CRDTs programmers are bound to two data structures: lists and maps. These data structures are not suited to implement an efficient tree data structure, i.e. one which provides logarithmic time lookups, insertions and deletions. Hence, JSON CRDTs are not general-purpose enough to implement an efficient tree-based text editor.

The quantitative evaluation focuses on measuring the performance of SECROs and JSON CRDTs with respect to a number of parameters: memory usage, time complexity, and throughput.

We found that SECROs efficiently manage memory, whereas the memory usage of (JSON) CRDTs grows unbounded. The time complexity benchmarks revealed that SECROs induce a linear time overhead which is proportional to the size of the operation history. Commit has shown to be crucial in order to keep the execution time within acceptable bounds. Finally, the JSON CRDT text editor scales better than the SECRO text editor.

Regarding the tree-based text editor, our experiments revealed that the tree structure induces a certain copy overhead. To reap the benefits of the tree structure, operations should manipulate (i.e. insert or delete) entire words, sentences or even paragraphs. However, single character manipulations are too fine-grained.

To summarize, we demonstrated that general-purpose available data types offering strong eventual consistency can be designed but re-ordering the operations is costly. The costs can be mitigated by periodically committing the replicas. Deciding on the appropriate commit strategy depends on the application and needs to be fine-tuned by the programmer.

7.2 Contributions

The research conducted throughout this dissertation led to several contributions. We outline each contribution within its respective domains:

CScript Within the field of distributed programming, we contribute CScript, a distributed programming language with native support for availability and consistency. CScript’s innovation is a distributed object-oriented programming model that offers replicated objects for which the programmer can specify whether the object should be strongly consistent or available (and only eventually consistent).

SECROs In the context of available data types and consistency, we contribute SECROs, a new consistency protocol for the strong eventual consistency model. SECROs are a *general-purpose* data type for building available data structures. Since operations do not commute conflicts can occur. How to solve these conflicts depends on the behaviour that is expected from the application.

Programmers use state validators to translate the expected behaviour into a set of invariants. State validators are associated to the operations of a SECRO and validate its state. We propose two state validators: preconditions and postconditions. The former avoids operations from running on a corrupted state, whereas the latter ensures that concurrent operations lead to a correct state.

When facing concurrent operations, SECROs use the state validators to re-order the operations in a way that satisfies the declared invariants.

JavaScript JSON CRDT To compare SECROs with JSON CRDTs, we implemented the JSON CRDT data type in JavaScript, based on the formal semantics described by (Kleppmann & Beresford, 2017). We thus contribute an implementation of the JSON CRDT to the JavaScript community.

7.3 Limitations And Future Work

We see four directions for future work. First, CScript maintains only a single copy of strongly consistent objects and serializes all updates on that copy. This copy forms a single point of failure and may become a performance bottleneck. Future work should address these problems by replicating the object to all peers. To keep the copies strongly consistent a distributed

consensus algorithm (Lamport, 1998; Ongaro & Ousterhout, 2014) will be needed.

Second, CScript targets peer-to-peer applications that run on top of full mesh networks, i.e. nodes are transitively connected. Future work could explore the integration of CScript with client-server architectures like the web. This requires a centralized peer lookup infrastructure and a central message broker for the publish-subscribe mechanism (see Chapter 5).

Third, the SECRO benchmarks in Chapter 6 revealed that the copy time dominates the execution time of operations. We argued that copying the state is needed because JavaScript is a mutable language and operations are executed tentatively. To validate our argument, future work should implement SECROs in a purely functional language.

Finally, the evaluation revealed that SECROs must be committed periodically in order to keep the performance within acceptable bounds. Up till now, we committed SECROs at fixed intervals. This might not be an optimal solution. Future work could investigate new commit strategies. A possibility would be to commit replicas whenever all copies are online at the same time.

References

- Almeida, P. S., Shoker, A., & Baquero, C. (2015). Efficient State-based CRDTs by Delta-Mutation. In A. Bouajjani & H. Fauconnier (Eds.), *International Conference on Networked Systems* (pp. 62–76). Agadir, Morocco.
- Bennett, J. K. (1987). The Design and Implementation of Distributed Smalltalk. In N. Meyrowitz (Ed.), *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications* (pp. 318–330). Orlando, Florida, USA. doi: 10.1145/38765.38836
- Bernstein, P. A., Burckhardt, S., Bykov, S., Crooks, N., Faleiro, J. M., Kliot, G., . . . Thelin, J. (2017). Geo-distribution of Actor-based Services. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), 107:1–107:26. doi: 10.1145/3133931
- Bernstein, P. A., Hadzilacos, V., & Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
- Brewer, E. A. (2000). Towards Robust Distributed Systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing* (p. 7). New York, NY, USA. doi: 10.1145/343477.343502
- Burckhardt, S. (2014). Principles of Eventual Consistency. *Foundations and Trends in Programming Languages*, 1(1-2), 1–150. doi: 10.1561/25000000011
- Burckhardt, S., Leijen, D., Protzenko, J., & Fähndrich, M. (2015). Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In J. T. Boyland (Ed.), *29th European Conference on Object-Oriented Programming (ECOOP 2015)* (Vol. 37, pp. 568–590). Dagstuhl, Germany. doi: 10.4230/LIPIcs.ECOOP.2015.568
- Cachin, C., Guerraoui, R., & Rodrigues, L. (2011). *Introduction to Reliable and Secure Distributed Programming* (2nd ed.). Hanover, MA, USA: Springer-Verslag.
- Coulouris, G. F., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design* (5th ed.). London, England: Pearson Education.
- Davey, B. A., & Priestley, H. A. (2002). *Introduction to Lattices and Order*. Cambridge, England: Cambridge University Press.
- Eugster, P. T., Felber, P. A., Guerraoui, R., & Kermarrec, A.-M. (2003). The Many Faces of Publish/Subscribe. *ACM Computing Surveys (CSUR)*, 35(2), 114–131. doi: 10.1145/857076.857078
- Fischer, M. J., Lynch, N. A., & Paterson, M. S. (1985). Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*

- (*JACM*), 32(2), 374–382. doi: 10.1145/3149.214121
- Google. (2016). *Closure Library*. <https://developers.google.com/closure/library/>. (Retrieved on May 29, 2018)
- Google. (2018). *Share files from Google Drive*. <https://support.google.com/docs/answer/2494822>. (Retrieved on June 17, 2018)
- Jul, E., Levy, H., Hutchinson, N., & Black, A. (1988). Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems (TOCS)*, 6(1), 109–133. doi: 10.1145/35037.42182
- Kleppmann, M., & Beresford, A. R. (2017). A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10), 2733–2746.
- Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), 558–565. doi: 10.1145/359545.359563
- Lamport, L. (1998). The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), 133–169. doi: 10.1145/279227.279229
- Letia, M., Prego, N., & Shapiro, M. (2009). *CRDTs: Consistency without concurrency control* (Tech. Rep.). Rocquencourt, France: INRIA. (RR-6956)
- Liskov, B. (1988). Distributed programming in Argus. *Communications of the ACM*, 31(3), 300–312. doi: 10.1145/42392.42399
- Meiklejohn, C., & Van Roy, P. (2015). Lasp: A Language for Distributed, Coordination-free Programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming* (pp. 184–195). New York, NY, USA. doi: 10.1145/2790449.2790525
- Ongaro, D., & Ousterhout, J. (2014). In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (pp. 305–320). Berkeley, CA, USA.
- Roh, H.-G., Jeon, M., Kim, J.-S., & Lee, J. (2011). Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *Journal of Parallel and Distributed Computing*, 71(3), 354–368. doi: 10.1016/j.jpdc.2010.12.006
- Shapiro, M., Prego, N., Baquero, C., & Zawirski, M. (2011a). *A comprehensive study of Convergent and Commutative Replicated Data Types* (Unpublished doctoral dissertation). Inria–Centre Paris-Rocquencourt; INRIA.
- Shapiro, M., Prego, N., Baquero, C., & Zawirski, M. (2011b). Conflict-free Replicated Data Types. In X. Défago, F. Petit, & V. Villain

- (Eds.), *Symposium on Self-Stabilizing Systems* (pp. 386–400). Grenoble, France.
- Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms* (2nd ed.). Upper Saddle River, New Jersey, USA: Prentice-Hall.
- Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., & Hauser, C. H. (1995). Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In M. B. Jones (Ed.), *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (pp. 172–182). New York, NY, USA. doi: 10.1145/224056.224070
- Van Cutsem, T., Mostinckx, S., Gonzalez Boix, E., Dedecker, J., & De Meuter, W. (2007). AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad Hoc Networks. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science* (pp. 3–12). Iquique, Chile. doi: 10.1109/SCCC.2007.4
- Vogels, W. (2009). Eventually Consistent. *Communications of the ACM*, 52(1), 40–44. doi: 10.1145/1435417.1435432
- Waldo, J., Kendall, S. C., Wollrath, A., & Wyant, G. (1994). *A Note on Distributed Computing* (Tech. Rep.). Mountain View, CA, USA: Sun Microsystems, Inc.
- Wehrle, K., Götz, S., & Rieche, S. (2005). Distributed Hash Tables. In *Peer-to-Peer systems and applications* (pp. 79–93). Heidelberg, Berlin, Germany: Springer-Verlag.