

## Flexible CRDTS for a demanding world

*An open and systematic approach to CRDT development*

Bauwens, Jim

*Publication date:*  
2024

*License:*  
CC BY-ND

*Document Version:*  
Final published version

[Link to publication](#)

*Citation for published version (APA):*

Bauwens, J. (2024). *Flexible CRDTS for a demanding world: An open and systematic approach to CRDT development*. [PhD Thesis, Vrije Universiteit Brussel]. Crazy Copy Center Productions.

### Copyright

No part of this publication may be reproduced or transmitted in any form, without the prior written permission of the author(s) or other rights holders to whom publication rights have been transferred, unless permitted by a license attached to the publication (a Creative Commons license or other), or unless exceptions to copyright law apply.

### Take down policy

If you believe that this document infringes your copyright or other rights, please contact [openaccess@vub.be](mailto:openaccess@vub.be), with details of the nature of the infringement. We will investigate the claim and if justified, we will take the appropriate steps.

# FLEXIBLE CRDTs FOR A DEMANDING WORLD

An Open and Systematic Implementation  
Approach to CRDT Development

Jim Bauwens

*Dissertation submitted in fulfillment of the  
requirement for the degree of Doctor of Sciences*

August 23, 2024

Promotor:

Prof. Dr. Elisa González Boix, Vrije Universiteit Brussel, Belgium

Jury:

Prof. Dr. Bas Ketsman, Vrije Universiteit Brussel, Belgium (chair)  
Prof. Dr. Paul Van Eecke, Vrije Universiteit Brussel, Belgium (secretary)  
Prof. Dr. Bruno da Silva Gomes, Vrije Universiteit Brussel, Belgium  
Prof. Dr. Carlos Baquero, University of Porto, Portugal  
Prof. Dr. Annette Bieniusa, University of Kaiserslautern-Landau,  
Germany

Vrije Universiteit Brussel  
Faculty of Sciences and Bio-engineering Sciences  
Department of Computer Science  
Software Languages Lab

Printed by  
Crazy Copy Center Productions  
VUB Pleinlaan 2, 1050 Brussel  
Tel / fax : +32 2 629 33 44  
crazycopy@vub.be  
www.crazycopy.be

ISBN 9789464948394  
NUR 989  
THEMA UMB

The work in this dissertation has been funded by a PhD fellowship of the Research Foundation Flanders (FWO) - Project number 1SA5220N/1SA5222N (FWOSB90).

©2024 Jim Bauwens

All rights reserved. No part of this publication may be produced in any form by print, photoprint, microfilm, electronic, or any other means without permission from the author.

# Abstract

The increase in internet-connected devices, including smartphones and IoT systems, has changed the landscape of distributed systems. To ensure availability, data is often replicated between servers or systems at the edge using a local-first approach. This improves performance but introduces challenges as different copies of data need to be kept up-to-date. Conflict-free Replicated Data Types (CRDTs) can be used to ensure the convergence of replicated data, without the need for manual conflict resolution strategies. Replicas can diverge temporarily, but once updates stop they will eventually converge.

During our research, we identified four main challenges related to the use of CRDTs. First, memory management is complicated due to the accumulation of metadata tracking causality between operations. Secondly, there is limited support for dynamic networks, which is essential for local networks when peers can join and leave. Third, the reliance on Reliable Causal Broadcasting in many CRDT designs can lead to unnecessary delays. Operations will be buffered until all causal dependencies are met, even if those dependencies are not logically related. Fourth, existing support for the composition and nesting of CRDTs is inflexible and limited, forcing ad-hoc implementations.

This dissertation investigates and develops solutions to these concerns using Flec — a framework we developed for building eventually consistent applications — as a laboratory for experimenting with novel CRDT designs and techniques. Flec offers a modular approach for developing CRDTs through an open implementation that reifies the replication and convergence process.

To address the unbounded accumulation of metadata, we first introduce a technique that leverages communication acknowledgements to eagerly determine causal stability, enabling early metadata removal. We

then adapt this mechanism to support networks where new peers can dynamically join, and show how peers can obtain correct states asynchronously.

Secondly, we propose an extension to pure operation-based CRDTs that improves their responsiveness. By allowing pending operations stored in the RCB buffer to be partially applied before all causal dependencies have arrived, the reactivity of replicas is enhanced.

Finally, we propose Nested Pure Operation-Based CRDTs, a novel framework for building nested replicated data structures. Our approach allows CRDTs to support composition without semantic changes or structural limitations.

We validate our contributions by implementing a range of novel structured RDTs within Flec. Each chapter includes a validation section that substantiates our proposed techniques and extensions. The results demonstrate the effectiveness of our approach, which we believe forms a step forward in enabling the usage of CRDTs in a wide variety of applications.

# Samenvatting

De groei van apparaten verbonden met het internet, waaronder smartphones en IoT-systemen, heeft het landschap van gedistribueerde systemen veranderd. Om beschikbaarheid te garanderen, worden gegevens vaak gerepliceerd tussen servers of systemen aan de rand via een local-first benadering. Dit verbetert prestaties, maar brengt uitdagingen met zich mee omdat verschillende kopieën van gegevens up-to-date moeten worden gehouden. Conflict-free Replicated Data Types (CRDT's) kunnen worden gebruikt om de convergentie van gerepliceerde gegevens te garanderen. Replica's kunnen tijdelijk van elkaar verschillen, maar zodra updates stoppen zullen ze uiteindelijk convergeren.

We hebben tijdens ons onderzoek vier belangrijke uitdagingen geïdentificeerd met betrekking tot het gebruik van CRDTs. Ten eerste is geheugenbeheer gecompliceerd door de accumulatie van meta-data die causaliteit tussen operaties vastlegt. Ten tweede is er beperkte ondersteuning voor dynamische netwerken waar peers kunnen toetreden en vertrekken. Ten derde kan de afhankelijkheid op Reliable Causal Broadcasting (RCB) in veel CRDTs tot onnodige vertragingen leiden. Operaties worden gebufferd tot dat alle causale afhankelijkheden zijn toegepast, zelfs als die afhankelijkheden niet logisch gerelateerd zijn. Ten vierde is de bestaande ondersteuning voor het combineren van CRDTs inflexibel, waardoor ad-hoc implementaties nodig zijn.

Dit proefschrift onderzoekt oplossingen voor deze problemen met behulp van Flec - een framework die we hebben ontwikkeld voor het bouwen van eventueel consistente applicaties - als laboratorium voor het experimenteren met CRDT-ontwerpen en technieken. Flec biedt een modulaire aanpak voor het ontwikkelen van CRDT's door middel van een open implementatie.

Om de onbegrensde accumulatie van meta-data aan te pakken, introduceren we eerst een techniek die communicatiebevestigingen gebruikt om snel causale stabiliteit te bepalen, waardoor meta-data vroegtijdig verwijderd kan worden. Vervolgens wordt deze techniek aangepast om dynamische netwerken te ondersteunen en laten we zien hoe nieuwe peers asynchroon een correcte staat kunnen verkrijgen.

Ten tweede stellen we een uitbreiding voor op pure operation-based CRDT's die hun reactievermogen verbetert. Door toe te staan dat pending operaties die zijn opgeslagen in de RCB buffer gedeeltelijk kunnen worden toegepast voordat alle causale afhankelijkheden zijn gearriveerd, wordt de reactiviteit van replica's verbeterd.

Tenslotte stellen we geneste, pure operation-based CRDT's voor, een nieuw framework voor het bouwen van geneste, gerepliceerde structuren. Met onze aanpak kunnen CRDT's compositie ondersteunen zonder semantische veranderingen of structurele beperkingen.

We valideren onze bijdragen door een reeks gestructureerde RDT's te implementeren binnen Flec. Elk hoofdstuk bevat een validatiesectie die onze voorgestelde technieken onderbouwt en aantoont dat onze bijdragen het gebruik van CRDTs vereenvoudigt.

# Acknowledgements

With this dissertation, my PhD journey comes to an end. I want to express my deepest gratitude to my promotor, Prof. Elisa González Boix, for her unwavering guidance over the past years, starting with my master's thesis and continuing through my PhD. I am immensely grateful for the support you have provided, the patience you have shown, and for constantly encouraging me to grow into a better researcher.

I extend my sincere thanks to the jury members, Prof. Dr. Carlos Baquero, Prof. Dr. Annette Bieniusa, Prof. Dr. Bas Ketsman, Prof. Dr. Paul Van Eecke, and Prof. Dr. Bruno da Silva Gomes, for taking the time to read and evaluate this thesis. The feedback has allowed me to polish and deliver an improved dissertation.

Next, I would like to thank my colleagues with whom I've shared this journey. Matteo, I am grateful for the many years we faced the challenges at VUB together. From classmates to officemates, from good friends to great friends, from one pizza to another. I owe you infinite beers, and we should schedule another pizza evening soon!

Carlos, thank you for tolerating me during the toughest parts of my PhD, for making the office a fun and lively place, and for all the engaging discussions we had. Kevin, your insights during our many interesting discussions on SEC have made me a more critical and reflective researcher, for which I am thankful. Scull, I deeply appreciate your help in reviewing this dissertation and for sharing countless (perhaps too many) coffees during the writing phase! I'd like to thank all the other members and previous members from DisCo as well (Aäron, Dina, Isaac, Carmen), who have been part of this journey, offering help, discussions, and friendship over the years. Of course, there are many more at SOFT who have helped make my PhD journey a great experience, and I want to give my thanks to everyone at the lab.

I would not be where I am today without the unwavering support of my family. First and foremost, I want to thank my parents, who instilled in me the value of critical thinking and provided me with the opportunities to explore my passions. To my siblings — Manuel, Michal, Noë, Johnny, Tabitha, Miriam, Salome, Melody, Lydia, Stefan, Daniel, Ruben, Rebecca, and Deborah — and my siblings-in-law — Noemí, Elena, Jonathan, Olivier, and Lenny — I am deeply grateful for all the support, help, and love you have shown me over the years. A special note of thanks goes to my sisters Michal and Miriam, who guided me through the challenges of homeschooling; your assistance has been invaluable, and I will never forget the countless hours you dedicated to helping us all achieve our educational goals.

I would also like to give a shout-out to my good friends Dilara, Paola, Badre, and many others who have made my time in Brussels truly memorable. Your companionship has been a source of joy and support, and I am deeply grateful for your friendship.

Finally, I want to end my acknowledgements by thanking my love and partner, Natalia. Tu amor, apoyo y comprensión han sido fundamentales para llegar hasta aquí. Estoy profundamente agradecido por tenerte a mi lado y sé que todo es posible con tu ayuda incondicional. No podría haber deseado una mejor compañera en este viaje, y espero con impaciencia todo lo que el futuro nos reserva.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Replicated Systems . . . . .	2
1.2	Replicated Data Types . . . . .	3
1.3	Problem statement . . . . .	5
1.4	Our Approach . . . . .	7
1.5	Contributions . . . . .	7
1.5.1	Supporting Publications . . . . .	9
1.6	Dissertation Outline . . . . .	10
<b>2</b>	<b>Context and Motivation</b>	<b>13</b>
2.1	Consistency in Replicated Systems . . . . .	13
2.1.1	Strong Consistency Models . . . . .	14
2.1.2	Weak Consistency Models . . . . .	15
2.2	Conflict-Free Replicated Data Types . . . . .	16
2.2.1	Concurrency Semantics and Propagation Protocols . . . . .	17
2.3	Pure Operation-Based CRDTs . . . . .	24
2.3.1	Replication and Memory Management . . . . .	25
2.3.2	A Pure-Operation Based Set . . . . .	26
2.4	State of the Art of CRDT Implementations . . . . .	29
2.4.1	Approaches to Replication and Distribution . . . . .	29
2.4.2	Memory Management in CRDTs . . . . .	32
2.4.3	Composition and Nesting in CRDTs . . . . .	33
2.5	Conclusion . . . . .	35
<b>3</b>	<b>Flec: a Programming Framework for Eventually Consistent Systems</b>	<b>37</b>

3.1	The Flec Programming Framework . . . . .	38
3.2	Ambient-Oriented Programming Programming in TSAT . .	39
3.2.1	Concurrent Programming . . . . .	40
3.2.2	Chainable Promises in TypeScript and TSAT . . . .	42
3.2.3	Distributed Programming . . . . .	44
3.2.4	Networking and Communication Channels . . . . .	48
3.3	CRDT and Replication in Flec . . . . .	52
3.3.1	Core Replication Algorithm . . . . .	52
3.3.2	Operation-Based CRDT Layer . . . . .	57
3.3.3	Pure Operation-Based CRDT API . . . . .	61
3.4	Conclusion . . . . .	64
<b>4</b>	<b>Efficiently Supporting Dynamic Networks</b>	<b>67</b>
4.1	Definitions and Assumptions . . . . .	68
4.2	A Dynamic Join Model for CRDTs . . . . .	69
4.2.1	Concurrent Joins . . . . .	70
4.2.2	Dynamic Join Algorithm . . . . .	72
4.2.3	Implementing our Dynamic Join Model in Flec . . .	75
4.3	Improved Metadata Removal with Eager Stability Determination . . . . .	75
4.3.1	Eager Stability in Dynamic Environments . . . . .	77
4.4	Implementing Eager Stability Determination in Flec . . . .	79
4.4.1	Extending the Replication Mechanism . . . . .	79
4.5	Evaluation of Eager Stability Metadata Removal . . . . .	82
4.5.1	Experiments . . . . .	82
4.5.2	Methodology . . . . .	83
4.5.3	Assessing Meta-Data Removal for the Pure Op-Based Framework without Eager Stability . . . . .	83
4.5.4	Assessing the Benefits of Stability Messages . . . . .	86
4.5.5	Assessing the Network Overhead . . . . .	88
4.5.6	Assessing the Benefits of Using Log Size as a Heuristic for Stability Messages . . . . .	89
4.6	Notes on Related Work . . . . .	91
4.7	Conclusion . . . . .	92
<b>5</b>	<b>Improving the Reactivity of CRDTs</b>	<b>95</b>

5.1	The Need for Reactive CRDTs . . . . .	96
5.2	Improving the Reactivity of Pure Operations-Based CRDTs	98
5.2.1	Extending Semantic Log Compaction with RPM- Log Support . . . . .	99
5.2.2	Reactive Pure Operation-Based Sets . . . . .	100
5.3	Implementation in Flec . . . . .	103
5.3.1	Implementing Reactive Sets in Flec . . . . .	104
5.4	Validation . . . . .	108
5.4.1	Experiments . . . . .	108
5.4.2	Assessing the Overhead of Our Approach . . . . .	108
5.4.3	Assessing Improved Reactivity with Our Approach .	109
5.5	Conclusion . . . . .	110
<b>6</b>	<b>Nestable Pure Operation-Based CRDTs</b>	<b>113</b>
6.1	Nesting Pure Operation-Based CRDTs . . . . .	114
6.1.1	Extending the Pure Operation-Based Framework . .	115
6.1.2	Formalised Semantics for Extended Functionality . .	120
6.1.3	Nested Pure Operation-Based Maps . . . . .	122
6.1.4	Discussion . . . . .	124
6.2	Implementation . . . . .	125
6.2.1	Implementing Nested CRDTs in Flec . . . . .	125
6.3	Validation . . . . .	129
6.3.1	Verification with VeriF <sub>x</sub> . . . . .	129
6.3.2	Portfolio of Nested CRDTs in Flec . . . . .	132
6.3.3	Use-Case: A Mixed CRDT-Based Distributed Filesystem . . . . .	133
6.3.4	Evaluation of Network Traffic in Comparison With Automerge . . . . .	135
6.4	Notes on Related Work . . . . .	140
6.5	Conclusion . . . . .	140
<b>7</b>	<b>Conclusion</b>	<b>143</b>
7.1	Restating Our Approach . . . . .	143
7.2	Contributions . . . . .	145
7.2.1	Technical Contributions . . . . .	146
7.3	Discussion . . . . .	146

7.3.1	Dynamic Join Model . . . . .	146
7.3.2	Eager Causal Stability . . . . .	147
7.3.3	Nested Pure Operation-Based CRDTs . . . . .	147
7.4	Future Work . . . . .	148
7.4.1	Reactive CRDTs . . . . .	148
7.4.2	Automatic Generation of Redundancy Relations . . . . .	148
7.4.3	Distributed Garbage Collection through Consensus . . . . .	149
7.4.4	Multi-Log Pure Operation-Based CRDTs . . . . .	149
7.4.5	Smart Concurrency Semantics Through NLP . . . . .	150
7.4.6	Comparison of Flec to Mainstream Frameworks . . . . .	150
7.5	Closing Remarks . . . . .	151
<b>A</b>	<b>DFS Code Listings</b>	<b>153</b>
<b>B</b>	<b>A Remove-Wins Pure Operation-Based CRDT</b>	<b>161</b>
<b>C</b>	<b>Implementation Details for Eager Stability Determination</b>	<b>163</b>

# Acronyms

**AMOP** Ambient Oriented Programming.

**API** Application Programming Interface.

**AW-Set** Add-Wins Set.

**CAP** Consistency, Availability, and Partition-tolerance.

**CRDT** Conflict-free Replicated Data Type.

**DFS** Distributed Filesystem.

**DSL** Domain Specific Language.

**EC** Eventual Consistency.

**JSON** JavaScript Object Notation.

**LWW** Last-Writer-Wins.

**MOP** Meta Object Protocol.

**MV-Register** Multi-Value Register.

**OR-Set** Observed-Removed Set.

**P2P** Peer-to-Peer.

**PO-Log** Partially Ordered Log.

**RCB** Reliable Causal Broadcasting.

**RDT** Replicated Data Type.

**RPM-Log** Reified-Pending-Messages Log.

**RW-Set** Remove-Wins Set.

**SEC** Strong Eventual Consistency.

# List of Figures

2.1	A representation of the state-based CRDT replication process.	18
2.2	A representation of the operation-based CRDT replication process. . . . .	21
2.3	The internal state of an AW-Set. One operation is causally stable and, as such, does not contain a timestamp. Together, the operations form the state $\{A,B,C\}$ . . . . .	26
2.4	The internal states of an AW-Set, after receiving a remove (rem) operation, and after the operation has been applied. .	28
3.1	Flec architectural overview. . . . .	38
3.2	Far reference can point to objects on local or remote actors.	45
4.1	Step 1: A node requests to join a network. . . . .	69
4.2	Step 2: The new node contacts all other nodes in the network.	70
4.3	Step 3: The new node performs a state request once it has been fully acknowledged. . . . .	71
4.4	Two nodes perform simultaneously join requests to different nodes in the network. . . . .	71
4.5	Node A forwards a link message from node O to node N. . .	71
4.6	Acknowledgements used by the RCB layer to ensure reliable delivery. . . . .	78
4.7	Letting other replicas know that an operation is stable. . .	78
4.8	Numbers of entries in log of a pure-operation based Remove-Wins set, as operations are being applied to the sets in the system. For every 100 operations, the source replica is changed. Measurements taken for a system with 2, 4, and 8 replicas. . . . .	85

4.9	Numbers of entries in log of a pure-operation based Remove-Wins set, as operations are being applied to the sets in the system. For every 100 operations, the source node is changed. Measured in a system with 4 replicas. The colours represent the source of the entries in the logs. . . . .	85
4.10	Comparison of the number of entries in the log of a pure operation-based Remove-Wins set in a system of 4 replicas, with no additional stability messages, stability messages every 10 operations, and stability messages every 50 operations. For every 100 operations, the source node is changed.	87
4.11	Detailed view at the number of entries in log of a pure-operation based Remove-Wins set in a system with 4 replicas and stability messages every 50 operations. . . . .	87
4.12	Same comparison as in Figure 4.10, but with the difference that the source node is changed every 200 operations. . . .	88
4.13	Comparison of the total number of sent messages for a pure-operation based Remove-Wins set in a system of 4 replicas, but with no additional stability messages, stability messages every 10 operations, and stability messages every 50 operations. For every 100 operations, the source node is changed. . . . .	89
4.14	Comparison of the number of additional messages sent in a pure operation-based Remove-Wins set across a system of 8 replicas, evaluating the impact of stability messages sent every 10 operations and every 50 operations against a baseline with no stability messages. The source node is changed every 100 operations. . . . .	90
4.15	Same comparison as in Figure 4.10, but with the difference that nodes will additionally try to trigger stability messages if the number of entries log exceeds a certain size (75 entries for the system with int=50 and 15 entries for the system with int=10). For every 100 operations, the source node is changed. . . . .	91

4.16	A detailed look at the number of entries in the log of a pure operation-based Remove-Wins set, in a system with 4 replicas and stability messages every 50 operations and when the number of entries in the log exceeds 75. For every 100 operations, the source node is changed. . . . .	92
5.1	Network connectivity between set replicas. . . . .	96
5.2	Log size of replica B over time. Measurements taken for a system with reactive AW-Set CRDTs and standard (non-reactive) AW-Set CRDTs. There is no delay between replicas A and B. . . . .	109
5.3	Log size of replica B over time. Measurements taken for a system with reactive AW-Set CRDTs and standard (non-reactive) AW-Set CRDTs. Operations between replicas A and B are delayed by 5 seconds. . . . .	110
6.1	Three stages of the internal state of a hash-map with update-wins semantics containing nested Multi-Value registers: 1) initial state, 2) arrival of an update (upd) operation, and 3) final state after applying the operation. . . . .	117
6.2	Example of a nested redundancy relation that selectively resets nested children, triggered by the deletion of a key. As the arriving delete (del) operation is concurrent with an update (upd) that arrived earlier, the nested child needs to be partially reset. . . . .	119
6.3	Network traffic (in bytes/op) originating from the source node for both Automerge and Flec. In every operation, a file is created and written. . . . .	136
6.4	Total cumulative networking traffic (in bytes/op) from all nodes for Automerge. In every operation, a file is created and written. . . . .	136
6.5	Network traffic (in bytes/op) originating from the source node for both Automerge and Flec. Every operation creates a new user, a new group, and a new file. The user is added to the group, and the file is created with the new user as the owner. Finally, the file is written. . . . .	138

6.6	Total network traffic (in bytes/op) for both Automerge and Flec. Every operation creates a new user, a new group, and a new file. The user is added to the group, and the file is created with the new user as the owner. Finally, the file is written. . . . .	138
6.7	Total network traffic for Automerge for the previous experiment, highlighting an issue with exponential growth after a certain number of operations. . . . .	139

# List of Tables

2.1	Semantics for the add-wins pure-op set, based on the approach in [BAS17]. . . . .	27
2.2	Semantics for the remove-wins pure-op set, based on approaches from [BGB20b, BAS17]. . . . .	29
3.1	Channel Types. . . . .	49
3.2	Core Channel Interface. . . . .	50
3.3	Core CRDT Interface in Flec. . . . .	53
3.4	Operation-Based CRDT Interface in Flec. . . . .	58
3.5	Pure Operation-Based CRDT Interface in Flec. . . . .	62
3.6	PO-Log Entry API in Flec. . . . .	63
4.1	Dynamic Network Hooks for the Operation-Based CRDT API in Flec. . . . .	76
5.1	A sequence of operations applied to the three add-wins pure operation-based set replicas. <code>Replica X :: Op(y)</code> denotes the application of an operation <code>Op</code> with arguments <code>y</code> on <code>Replica X</code> . The last remove does not have any immediate effect on A, as A is waiting for B before it will apply any other operation from C. . . . .	97
5.2	A sequence of operations applied to the three classic OR-Set replicas. <code>Replica X :: Op(y)</code> denotes the application of an operation <code>Op</code> with arguments <code>y</code> on <code>Replica X</code> . The last remove is applied immediately on set A. . . . .	97
5.3	Modified semantics for the Add-Wins pure-op set, supporting RPM-Log (based on approach in [BAS17]). . . . .	101

5.4	Modified semantics for the RW-Wins pure-op set, supporting RPM-Log (based on approach in [BGB20b, BAS17]). . .	102
5.5	Reactive Pure Operation-Based CRDT API in Flec. . . . .	103
6.1	Update-wins pure operation-based map, with support for nested CRDTs. . . . .	123
6.2	Remove-wins pure operation-based map, with support for nested CRDTs. . . . .	123
6.3	Nesting Operation-Based CRDT Interface in Flec. . . . .	126
6.4	Implemented nested CRDT types. . . . .	133
A.1	Legend for the TypeScript classes and types used in the DFS implementation. . . . .	153

# Listings

3.1	Code snippet of a 'Ping Pong' application showing actor creation in TSAT. . . . .	41
3.2	Code snippet of a 'Ping Pong' application that shows asynchronous message sends. . . . .	41
3.3	Code snippet demonstrating promise chaining. . . . .	43
3.4	Code snippet demonstrating promise chaining combined with an await statement. . . . .	44
3.5	Code snippet showing a TSAT actor exporting a PingPong object on the network. . . . .	46
3.6	Code snippet showing a TSAT actor discovering a PingPong object and sending a message to its far reference. . . .	46
3.7	PingPong Class. . . . .	47
3.8	Code snippet showing MQTT configuration in TSAT . . . .	49
3.9	Code snippet showing the use of a PN-Counter CRDT in Flec. . . . .	55
3.10	Code snippet showing the implementation of a state-based PN-Counter CRDT in Flec. . . . .	56
3.11	Code snippet showing the implementation of an operation-based counter CRDT in Flec. . . . .	60
3.12	Pure operation-based AW-Set implementation. . . . .	65
3.13	SetOperation Type Interface. . . . .	66
5.1	A code extension to the pure op-based framework in Flec that enables reification of buffer data. . . . .	104
5.2	Reactive AW-Set implementation in Flec. . . . .	105
5.3	Reactive RW-Set implementation in Flec. . . . .	106
6.1	The implementation of an RW-Map in Flec, using the described extensions (A). . . . .	127

6.2	The implementation of an RW-Map in Flec, using the described extensions (B).	128
6.3	Convergence update-update.	130
6.4	Convergence update-delete.	131
6.5	Convergence delete-delete.	131
A.1	Access right class representations.	154
A.2	DFS CRDT Interface.	155
A.3	The general structure of the DFS nested CRDT, highlighting the main nested children that contain the filesystem meta-data.	156
A.4	Structure of the operation handling code for the DFS. Included is the code for the CreateFile callback, which can either be invoked locally or as a result of a replicated operation.	158
A.5	User API for local mutations to DFS CRDT, allowing simple modification of the DFS meta-data.	159
A.6	Example test code for the DFS CRDT, which creates a new admin user, a new group, adds the user to a group, and then creates and writes a file with this new user.	159
B.1	RW-Set implementation in Flec.	162
C.1	Structure of the PureOpCRDT class, used to implement pure-operation based CRDTs.	164
C.2	The onOperation method is used to process received operations.	165
C.3	Methods allowing instrumentation of the cleanup process.	166
C.4	The logic used to mark and compact causally stable log entries.	167

# Chapter 1

## Introduction

Since the advent of computer networks in the late 1960s, distributed programming has become a large field within computer science. Distributed computing allows the execution of complex tasks by using the collective power of different computing machines. Over the years, the importance of distributed programming has increased significantly as the number and variety of interconnected devices have exploded due to advancements in Internet technologies, cloud computing, and the miniaturisation of hardware.

With the increase in interconnected devices, we have also witnessed a large increase in the volume of data generated and shared among distributed systems. This has created a demand for robust and scalable distributed systems that manage data across various locations without sacrificing performance. Modern distributed applications range from collaborative word processing software to decentralised data management systems that ensure data integrity and consistency across global networks.

This dissertation focuses on programming support for developing robust and distributed systems. These systems have challenges and complexities that do not exist in non-distributed environments, such as the need for coordination mechanisms to prevent conflicts and ensure smooth operation.

## 1.1 Replicated Systems

Distributed systems often rely on data replication to enhance reliability, availability, and performance by duplicating data across multiple nodes. By maintaining multiple copies of data or services, replicated systems can provide high availability and fault tolerance, as any individual node's failure does not compromise the overall system's functionality. Replication can also lower system latency, as data can be placed closer to clients, improving access times.

Despite their advantages, replicated systems are challenging to design as data needs to be kept consistent. Without proper mechanisms to synchronise replicas, discrepancies can arise, leading to unreliable system behaviour. Different strategies exist and can range from strong consistency models, where all replicas reflect the same state at all times, to eventual consistency models, where replicas may temporarily diverge but are guaranteed to converge over time. The CAP theorem [Bre00, GL02] provides a framework for understanding the trade-offs that different strategies may bring.

**The CAP Theorem.** CAP stands for *Consistency, Availability, and Partition Tolerance*, and the theorem states that a distributed system can only simultaneously achieve two out of these three properties. The theorem defines the three properties as follows:

- Consistency (C): Every read receives the most recent write or an error.
- Availability (A): Every request receives a response (non-error), regardless of the system state.
- Partition Tolerance (P): The system continues to operate despite arbitrary partitioning due to network failures.

Given these constraints, designers of replicated systems must make trade-offs based on the specific requirements and characteristics of their applications. For instance, a distributed system handling financial transactions may prioritise consistency and partition tolerance, ensuring that all nodes reflect the latest data, even if it means sacrificing availability

during network partitions. Conversely, social media platforms may prioritise availability and partition tolerance, allowing users to continue posting and reading updates (such as tweets and threads) even if some parts of the platform are not synchronised; e.g., it is perfectly fine that different users observe updates in different orders. In practice, any modern application needs to be partition tolerant, and as such, developers will need to choose between consistency and availability.

## 1.2 Replicated Data Types

Early work in distributed systems focused on ensuring (strong) consistency by sacrificing availability through consistency protocols [Lam78, Gra78, OO14]. However, work such as Bayou [TTP<sup>+</sup>95] and Dynamo [DHJ<sup>+</sup>07] showed that it was possible to build scalable fault-tolerant distributed systems guaranteeing weaker forms of consistency, such as eventual consistency (EC).

Bayou introduced programming abstractions that allowed developers to determine when conflicts occurred during concurrent updates and provided constructs to deal with such problems. Dynamo took an approach where all modifications were always accepted; however, the developers had to deal with potential conflicts when accessing data, either with basic strategies provided by Dynamo (e.g., last-writer-wins) or by manually encoding a conflict resolution strategy.

Operational Transformation (OT) [EG89] was an alternative approach explored to ensure state convergence in replicated systems, and EC as an extension. The approach used transformation functions to modify arriving operations so that their intended effects would be preserved without conflicts when applied together with other concurrent operations. However, many OT mechanisms have since been proven incorrect [DPFGB23] and, as such, the approach has mostly been abandoned.

Recent work investigates high-level replicated data types (RDTs), data structures that are replicated across devices, such as Conflict-free Replicated Data Types (CRDTs) [SPBZ11a] and cloud types [BFLW12]. CRDTs are of special interest because they support an extension to EC called Strong Eventual Consistency (SEC), where a system is defined so that conflicts cannot arise. SEC adds the guarantee of *state convergence* to EC, i.e. if two replicas of the data type have received the same updates, they

will be in the same state. As a result, developers do not need to implement custom conflict resolution schemes, and no synchronisation is needed to resolve conflicts, yielding lower latency and better performance/scalability [KPSJ19, Sha17, Pre18].

In this work, we investigate implementation aspects for CRDTs, and as such, we will give a brief overview of the historical context.

**Early Adoption of CRDTs.** Before their formal introduction, CRDTs were already explored in various forms, although under different names. Baquero et. al [BM99] explored replicated data types which were essentially state-based CRDTs. Treedoc [PMSL09] was introduced as a Commutative Replicated Data Type, and the approach follows the design of Operation-Based CRDTs. The authors of these works realised the relation between the different works and together introduced Conflict-free Replicated Data-Types in the 2011 comprehensive study [SPBZ11a].

Early on, Riak [Klo10] emerged as one of the first commercial adopters, and incorporated CRDTs into their distributed key-value store. Several CRDT designs were also integrated into the Akka framework as part of its distributed data features. The work in both of these platforms enabled developers to build highly available geo-distributed applications.

In parallel, research projects such as the SyncFree European Project led to the development of novel geo-replicated databases such as AntidoteDB [Ant]. AntidoteDB incorporated novel techniques that allowed it to work more efficiently than previous solutions, and it has since become an incubator for new CRDT research.

However, most of the focus then was on devising correct specifications for CRDTs. While some of these designs were picked up by different systems, they were mostly implemented in an ad-hoc fashion. This limited the availability of CRDT to other platforms, as the approaches were not designed with portability in mind.

Around 2018, a paradigm shift formed, where user-centric collaborative applications gained attention. In collaborative applications, multiple users may work together on the same document, and as such, concurrent operations are common. CRDTs proved to be a natural use-case for such environments, and work such as JSON CRDTs [KB17] were ideal for the complexity of collaborative environments.

**A push for Local-First.** Following the foundational work with JSON CRDTs, Automerge was released as a library designed to enable real-time collaboration across applications where users can modify data concurrently and offline, with changes automatically merging without conflicts. Together with other libraries such as Yjs [Yjs], a new *Local-first* software principle was born, with CRDTs as a technical foundation.

Local-first software [KWvHM19] forms a set of principles for software for giving users back the ownership of their data by allowing its offline storage. The core idea is that users should be able to work on their data without the need for communication through external servers and be able to collaborate across multiple devices. In general, the principles aim to improve data security, privacy, and long-term preservation. Local-first software has seen a large surge in interest in the industry over the past few years, with new software implementations and frameworks integrating the paradigm, such as ElectricSQL [Ele] and CosmosDB [Cos].

### 1.3 Problem statement

We observe that the adoption of CRDTs in distributed systems is still limited by several challenges. Current CRDT designs are not universally applicable; they often struggle to integrate with existing systems and platforms and rely on ad-hoc or rigid approaches for replication, memory management, and composition.

In this dissertation, we study programming language implementation techniques for systematically handling the aforementioned concerns in CRDTs. Our work aims to decrease the hurdles developers face before they can extend or even implement existing solutions. We categorise the identified issues into three categories:

**Replication and distribution concerns.** CRDTs are replicated data structures and, as such, have to deal with typical distribution problems, such as concurrency, partial failures, and the lack of global clocks. CRDT designs are typically expressed as specifications, making certain assumptions about the environment they are used in. Often, however, these assumptions may not reflect reality, or key information on how to implement them into software is lacking. For example, some designs assume a fixed number of replicas, blocking them from being used in dynamic environments where the number

of nodes is not fixed. In other cases, CRDTs require causal delivery guarantees, which may impose a too high burden on devices with limited memory resources. Concretely, implementors must develop their own techniques to deal with various replication and distribution aspects because no systematic approaches exist to handle these concerns.

**Memory management.** Due to the concurrent nature of CRDTs, keeping track of the order and causality of operations is essential. This is typically done through causal metadata, such as vector clocks or unique identifiers. If not managed properly, this metadata can build up in a system, leading to memory resource problems. Due to the distributed nature of systems, no shared memory space can be garbage-collected, and memory management becomes non-trivial.

Existing work [BZP<sup>+</sup>12, BAS14, Pre18, RJKL11] explores how CRDTs can avoid metadata or how it can be removed through garbage collection mechanisms. For example, the pure operation-based CRDT framework [BAS17] provides a systematic way to minimise resource usage through *redundancy relations*. The approach, however, relies on a causal delivery middleware for all operation delivery. While this simplifies design, it may introduce unneeded delays in operation application (i.e. making the CRDTs less reactive). Additionally, the pure operation-based CRDT specifications do not detail how it should be implemented, leaving implementors to deal with the replication and distribution concerns.

**Composition and nesting.** Most applications rely on complex, structured data for their operations. For example, in a collaborative drawing tool, the replicated state may need to contain a list of drawn objects, along with the properties of each of these objects. However, the composition of CRDTs is a largely unexplored and non-trivial problem. The main challenge is ensuring that SEC is upheld within nested structures, as the convergence mechanisms used for singular CRDTs are not designed for composition [SPBZ11a, Pre18, WMM20, WQK<sup>+</sup>23]. Current approaches focus mainly on static composition (where the structure of replicated objects cannot be changed at runtime) or limit composition to predefined CRDT types [KB17, NJDK16, LSB<sup>+</sup>19]. Without support for custom CRDTs,

creating complex structures tailored to application requirements becomes impossible.

As such, we identify that there is no systematic approach for creating dynamically nestable structures with general-purpose CRDTs.

## 1.4 Our Approach

In this dissertation, we devise and implement a CRDT framework that addresses development concerns caused by the aforementioned problems. Our approach to ease the construction, design, and implementation of CRDTs follows two main ideas:

**A Systematic Approach to CRDT Designs.** Many innovative CRDT designs are hampered through an ad-hoc design approach, where the designs cannot easily be combined with existing techniques, and as such, implementing solutions for the aforementioned problems becomes a complicated task. We aim to explore a systematic approach for dealing with concerns such as memory management, reactivity, composition and nesting, and more. The goal is to provide a framework for abstracting these details away from individual data types so that they do not need to be reimplemented per data type.

**An Open Implementation Approach to CRDT Development.**

Inspired by open implementations such as CLOS [GWB91], where developers can modify language behaviour from within the language, we aim to design a CRDT middleware as an open implementation to test solutions and design extensions for CRDTs. By opening up the core replication framework of the middleware, we can provide flexible interfaces that can be used to modify and tweak the capabilities of CRDT designs. We aim to allow our implementations to closely follow formal specifications and be ported to other frameworks and systems. It should also simplify the experimentation of novel constructions and extensions, opening the exploration of different design choices.

## 1.5 Contributions

This dissertation presents the following main contributions:

**Eager Causal Stability in CRDTs.** We introduce a novel mechanism to eagerly determine causal stability, improving causal metadata removal. Our approach allows for a faster metadata cleanup process and allows systems to have a lower memory consumption. Additionally, application implementors can fine-tune these processes, allowing for a balance between network resource usage and memory consumption in their systems.

**A Join Model for CRDTs in Dynamic Networks.** We study the challenges of employing CRDTs in dynamic network environments and propose a join model to support dynamic networks where new nodes can join at any moment. Our approach ensures new nodes can acquire a correct replication state, allowing them to participate effectively in the replicated system.

**Improved Reactivity for CRDTs.** We study the challenges introduced by the enforcement of causal ordering in many CRDT designs. We show that causal ordering may lead to less reactive CRDTs, as operations may be delayed needlessly. We propose an approach to improve the reactivity of frameworks relying on causal ordering through the reification of the buffered operations. In particular, we apply it as an extension to the pure operation-based CRDT framework and evaluate its effectiveness.

**Nestable Pure Operation-Based CRDTs.** We explored a structured approach for designing nested CRDTs based on the pure operation-based CRDT framework. We introduce nested pure operation-based CRDTs and show how several common nested data structures can be designed and modelled in the framework. To support our work, we validate our approach by extending our pure operation-based framework in Flec to include support for nested pure operation-based CRDTs and implementing a portfolio of commonly nested data structures.

To support our research, we developed an artefact named Flec, which we present as our technical contribution. The Flec programming framework facilitates the implementation of distributed programming with CRDTs. By integrating the advantages of TypeScript and the Ambient-Oriented programming model [DVCM<sup>+</sup>06, CMGB<sup>+</sup>07], Flec provides a versatile solution for addressing the complexities inherent in distributed computing.

Flec offers an open, extensible programming interface for implementing CRDTs, which allows it to be used as a laboratory for experimenting with CRDTs. Throughout the next chapters, Flec is used to implement, test, and validate our conceptual contributions.

### 1.5.1 Supporting Publications

- **Flec: a versatile programming framework for eventually consistent systems.** Jim Bauwens and Elisa Gonzalez Boix. 7th Workshop on Principles and Practice of Consistency for Distributed Data - PaPoC '20 (April 2020) [BGB20a]

This paper introduces the initial version of Flec, our open framework for implementing and experimenting with CRDTs in TypeScript. We detail the framework's extensibility and explain how it can be used in various system settings. In Chapter 3, we describe an extended version of the original Flec, which forms the basis for all of our implementations.

- **Memory Efficient CRDTs in Dynamic Environments.** Jim Bauwens and Elisa Gonzalez Boix. 11th International Workshop on Virtual Machines and Intermediate Languages - VMIL '19 (October 2019) [BGB19]

This workshop paper introduces our initial work on dynamic networks. It introduces the dynamic join model and details how garbage collection can be improved in such environments. The concepts of this paper are the basis for Chapter 4.

- **From Causality to Stability: Understanding and Reducing Meta-Data in CRDTs.** Jim Bauwens and Elisa Gonzalez Boix. 17th International Conference on Managed Programming Languages and Runtimes - MPLR '20 (November 2020) [BGB20b]

Extending on our workshop paper, this publication brings an improved design for the memory management approach in dynamic networks. It provides an extended evaluation through implementation in Flec, which we also detail in Chapter 4.

- **Improving the Reactivity of Pure Operation-Based CRDTs.** Jim Bauwens and Elisa Gonzalez Boix. 8th Workshop on Principles and Practice of Consistency for Distributed Data (April 2021) [BGB21]

This paper explores problems related to CRDT reactivity and proposes a solution by extending the pure operation-based CRDT framework. The extension reifies the causal RCB buffer and allows implementors to reason about the effects of buffered operations through novel redundancy relations. We provide an implementation of this work in Flec, validate how it can improve reactivity, and detail how improved reactivity leads to memory management improvements. This work provides the core for Chapter 5.

- **Nested Pure Operation-Based CRDTs.** Jim Bauwens and Elisa Gonzalez Boix. In 37th European Conference on Object-Oriented Programming (ECOOP 2023). Leibniz International Proceedings in Informatics (LIPIcs), Volume 263, pp. 2:1-2:26, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2023) [BGB23]

This publication describes our approach to nested CRDTs by formally extending the pure operation-based CRDT framework. The paper validates the extension through a verified implementation in VeriF<sub>x</sub>. We additionally provide a full-fledged implementation in Flec and a portfolio of nested data types. This work provides the core for Chapter 6.

## 1.6 Dissertation Outline

**Chapter 2: Context and Motivation.** This chapter provides the context for our work, starting with the description of the most relevant consistency models. We explain the different tradeoffs between strong and weak models and end with a focus on *Strong Eventual Consistency*. We then move to CRDTs, where we look at the different families, and explore in what cases can be used. Finally, we give an overview of the state-of-the-art in CRDT design and implementations, and what current solutions may exist that address concerns from the aforementioned problems.

**Chapter 3: Flec: a Programming Framework for EC Systems.**

This chapter explores our technical contribution, Flec. We detail how it can be used for building concurrent and distributed programming and explain how it adheres to the Ambient-Oriented programming paradigm. We then explore Flec’s CRDT framework and how CRDTs can be implemented and used in the framework. We detail in full the extensible API, which will be used to implement the designs introduced in later chapters.

**Chapter 4: A Dynamic Join Model for CRDTs.** This chapter looks into our dynamic join model and the eager causal stability extensions. We explain the need for such improvements and introduce our models. We provide algorithmic specifications for our approach and how these can be implemented in the Flec framework. We finish with an evaluation of the approach.

**Chapter 5: Improving the Reactivity of CRDTs.** This chapter discusses the downsides of using reliable causal broadcasting and how it may impact the reactivity of CRDTs. We introduce a systematic approach to improve the reactivity of CRDT designs and describe it as an extension of the pure operation-based CRDT framework.

**Chapter 6: Nestable Pure Operation-Based CRDTs.** This chapter introduces nested pure operation-based CRDTs, a systematic approach to the composition of CRDTs. We provide a formal description of the extension, and implement the design both in VeriFx and Flec. The VeriFx implementation is used to verify the correctness of our approach, and the Flec implementation is used to benchmark performance. The benchmarks show that our approach performs well when compared to a state-of-the-art framework.

**Chapter 7: Conclusion.** This chapter concludes this dissertation and provides an overview of our contributions and approach. We discuss the limitations and describe what future work may bring.



# Chapter 2

## Context and Motivation

In this chapter, we provide context for our research and motivate the need for systematic CRDT programming techniques. We first give an overview of several types of consistency that can be achieved in distributed systems, what requirements are needed to achieve different consistency levels, and how they are typically used. Following this, we zoom into Strong Eventual Consistency (SEC) and CRDTs. We overview the different strategies that can be used to design and implement such data types. Finally, we look at the state-of-the-art CRDT implementations and the difficulties developers might face when dealing with the problems we identified in Chapter 1.

### 2.1 Consistency in Replicated Systems

As mentioned in Section 1.1, data replication across distributed components is a common strategy serving two main purposes: ensuring data availability in the face of partial system failures and improving system performance, for example, by lowering latencies by placing data closer to users. When users access or modify information in a computer system, they expect it to provide up-to-date content and that successive accesses return data corresponding to previous modifications. For example, in banking systems, the withdrawal of money from an account is expected to be directly reflected in the balance of that account.

Ensuring replicas to be consistent in distributed and replicated systems is a complex task, mainly due to the possibility of concurrent updates to such systems. Consistency models precisely define what is and

is not expected, specifying the consistency expected between system components and the operations applied to those components. Tanenbaum [TvS06] defines consistency models as a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly. Normally, a process that performs a read operation on a data item, expects the operation to return a value that shows the results of the last write operation on that data. Systems designers may choose to use particular consistency models to ensure different guarantees in various parts of their system.

*Strong* consistency models ensure the highest form of guarantees in distributed systems but require synchronisation, which can induce high latencies. On the other hand, *Weaker* consistency models relax certain guarantees and do not have the same demanding system requirements. The following sections delve into various consistency models, their practical applications, and the level of consistency they can guarantee.

### 2.1.1 Strong Consistency Models

Strong consistency models guarantee a consistent view of all writes (data mutations) to a system. Any read (data access) must behave in the same way for any user on any part of the system, which is typically achieved by enforcing a total order for all events (such as reads and writes).

Serializability [BHG86] is a strong consistency model for database systems, where the execution of transactions must correspond to a specific serial ordering of those transactions. This means that all sets of concurrent transactions must be given a fixed order and executed in this order on every system component.

Linearizability [HW90] is another strong consistency model that guarantees that for each write, all subsequent read operations will reflect that update or any later ones. All writes must be executed atomically (i.e., they cannot be interrupted by other system operations), and the execution order for all operations must correspond to their real-time invocation.

Coordination is required among all components of a distributed system to achieve strong consistency. In a replicated system, this means synchronisation between all replicas through a protocol, e.g., such as the two-phase commit protocol typically employed in distributed transactions. This requirement may impact system availability in systems prone to fail-

ures or with a high number of replicas, corresponding to the CAP theorem (as defined in Chapter 1).

Systems where accuracy and data uniformity are critical will typically use strong consistency. For example, most financial systems, such as banking and stock trading platforms, rely on strong consistency to ensure that all transactions are correctly recorded and consistent across the system. This prevents issues such as double spending or inconsistent records.

### 2.1.2 Weak Consistency Models

Weak consistency is a family of relaxed consistency models that does not guarantee immediate visibility of updates to all processes. It allows some operations on replicated data to be seen out of order or delayed and, as a result, offers greater flexibility and performance benefits compared to stricter consistency models.

#### 2.1.2.1 Causal Consistency

Causal consistency is a weak consistency model that guarantees that all replicas must execute causally dependent operations in the same order. For instance, if a particular operation A happens before operation B, then this order must be maintained across all replicas.

This consistency model is useful in scenarios where a total ordering of all operations is unnecessary, but the sequence of causally related events needs to be respected. For example, social media platforms may not need a fixed total order for all events in their system, as many events are unrelated. Causal order, however, is necessary to ensure that, for example, post replies do not appear before their parent posts.

While causal consistency removes the need for coordination in a distributed system, enhancing system availability and performance as a result, it can lead to diverged replicas if concurrent operations are not given a fixed order. For this reason, additional logic is required to handle concurrent operations. Often, this is achieved by combining causal consistency with the *Eventual Consistency* model, described below.

### 2.1.2.2 Eventual Consistency

The Eventual Consistency (EC) model describes a weak form of consistency where the state of replicas in a distributed system can temporarily diverge. It guarantees that if no new updates are made to a system, eventually, all replicas will converge to the same state that reflects the latest updates [Vog08, DHJ<sup>+</sup>07]. Systems that implement the EC model can ensure high availability, as they do not need to wait for synchronisation to terminate before allowing operations to be applied. The model, however, does not specify how EC must be achieved. It be achieved through conflict resolution mechanisms or even by dropping all operations.

**Strong Eventual Consistency** Strong Eventual Consistency (SEC) [SPBZ11b] is an extension of Eventual Consistency, where EC is guaranteed through *strong convergence*. Strong convergence enforces that regardless of the order in which operations arrive, if a set of replicas have received the same operations, they must be in the same state. This implies that systems implementing SEC do not need to synchronise to converge; receiving operations are enough for all systems to be in the same state. SEC can be achieved by implementing a system that guarantees causal consistency and that all concurrent operations are commutable. CRDTs are examples of such implementations, and the following section will explore them in detail.

## 2.2 Conflict-Free Replicated Data Types

As highlighted by the introduction, this dissertation focuses on improving programming support for CRDT implementations. As such, we will now provide the background on CRDTs required to follow our contributions.

Conflict-Free Replicated Data Types (CRDTs) are a family of data structures that guarantee strong eventual consistency by ensuring conflict-freedom for all operations [SPBZ11a, Pre18]. The core idea of CRDTs is to provide a replicated data type that exhibits an API similar to that of a sequential data type while guaranteeing *eventual state convergence* under concurrent operations. Updates can be applied locally at a replica, and the changes will eventually be propagated to all other replicas. This means that the state of two replicas will eventually become equivalent when they

have received the same operations, regardless of the order in which they arrive.

### 2.2.1 Concurrency Semantics and Propagation Protocols

As mentioned before, replicas may receive operations in different orders. However, in the semantics of most sequential data structures, applying two operations in different orders can lead to different results.

For example, a set data structure traditionally supports `add` and `remove` operations. These operations do not commute, as can be demonstrated by simply comparing the two different possible orderings of an `add` and `remove` applied to an empty set:

1. `set <- add(X)`
2. `set <- remove(X)`

This will result in a set without item `X`. The following will result in a set with item `X`:

1. `set <- remove(X)`
2. `set <- add(X)`

As such, we cannot directly use normal sequential data structures as replicated data structures, as replicas might diverge when operations are received in different orders. A set CRDT implementation will need *concurrency semantics* to ensure that all replicas apply operations in the same order to obtain the same state. Concurrency semantics are used to specify how CRDTs should deal with concurrency. Since there are many ways to resolve concurrent operations, a sequential data structure can have various concurrent CRDT variants that implement different concurrency semantics. For example, replicated sets will typically be available with *add-wins* and *remove-wins* semantics.

There exist two large families of approaches to design CRDTs: *operation-based* and *state-based* [BM99, PMSL09, SPBZ11a, Pre18]. Operation-based approaches ensure that all potentially conflicting operations commute. Replicas propagate all locally applied operations to all other replicas, where they will be processed and applied to the state. State-based

approaches ensure that updates to a replica can only monotonically increase the state (as a join-semilattice). Replicas can share this state and merge it through a specially designed function. In the following sections, we describe both approaches in more detail and discuss the implications of their designs.

### 2.2.1.1 State-Based CRDTs

State-based CRDTs (or CvRDTs) are CRDTs where the state forms a join-semilattice, and local updates monotonically increase the state [SPBZ11a, Pre18]. Besides local updates, a state-based CRDT can also be updated by merging in states from other replicas. The merge operation of a state-based CRDT must be designed to be commutative, associative, and idempotent, ensuring that replicas converge to the same state regardless of the order in which updates and merges occur.

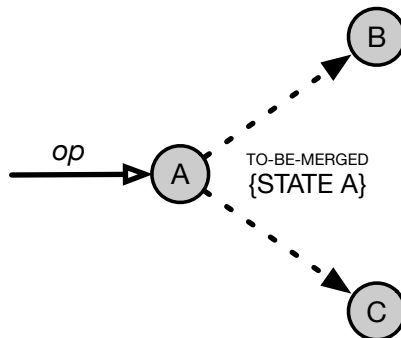


Figure 2.1: A representation of the state-based CRDT replication process.

Figure 2.1 shows the typical replication protocol in a system using state-based CRDTs. In the example, the system exists out of three replicas:  $A$ ,  $B$ , and  $C$ . An operation  $op$  is applied to replica  $A$ . At some point, replica  $A$  will broadcast its state to the other replicas. This state is the full state of  $A$ , containing the effects of the applied operation  $op$ , and any other operations that may have been applied previously. Replicas  $B$  and  $C$  will merge this state together with their own. They might also update the state concurrently and broadcast it to the other replicas at some other point.

Note that it is not critical if a broadcasted state fails to arrive at a replica (e.g., due to packet loss), as the next time it receives a state,

the new state will also contain the effects of the previous state update. The framework or middleware implementing state-based CRDTs has to decide when to broadcast state updates to other replicas. Depending on the system, the update rate must be fine-tuned to limit network resource usage.

**Delta-Based CRDTs.** A downside to state-based approaches is that replicas must broadcast the full application state. When the state grows to a certain size, state propagation may stress system and network resources. Delta-state CRDTs[ASB14, Pre18] are an optimised form of state-based CRDTs. Instead of sending the entire state during synchronisation, delta-state CRDTs only broadcast the changes or deltas. These deltas are the minimal information (e.g., the modified subset of the state) required to update other replicas about the changes made. This approach reduces the amount of data transferred during synchronisation, making it more efficient.

---

**Algorithm 1:** State-Based PN-Counter, based on design from [SPBZ11a].

---

```

statei:
  posCi(id) → number           ▷ i is the unique replica id
  negCi(id) → number           ▷ every unset id maps to 0
queryi getValue() : number
  | (∑(r,p)∈posCi p) - (∑(r,n)∈negCi n)
updatei inc(n: number)
  | posCi[i] := posCi[i] + n           ▷ n must be positive
updatei dec(n: number)
  | negCi[i] := negCi[i] + n           ▷ n must be positive
mergei (statej)
  | for (r, n) ∈ posCj do
  | | posCi[r] := max(posCi[r], n)
  | end
  | for (r, n) ∈ negCj do
  | | negCi[r] := max(negCi[r], n)
  | end

```

---

**A State-Based Counter CRDT.** To illustrate a state-based design, consider a state-based counter. Algorithm 1 shows the specification for a state-based Positive-Negative (PN) counter CRDT [SPBZ11a]. A PN-counter behaves similarly to a sequential counter data structure, representing a numeric value that can be incremented and decremented.

To achieve a mergeable state where concurrent increments (or decrements) do not get lost, increments and decrements are tracked for each replica. The current counter value for a replica can be computed by summing all increments and subtracting all decrements. Concretely, the state of a replica is defined through two maps<sup>1</sup>, one for tracking increments (*posC*) and one for decrements (*negC*), where the keys are replica IDs (we assume that each replica has a unique identifier). To increment or decrement the state of a replica, the update functions *inc* or *dec* can be used, respectively. For increments, the value in the *posC* map at key *i* will be incremented, where *i* is the ID of the updated replica. For decrements, the value in *negC* will be incremented.

When merging the state of two replicas, the *posC* and *negC* maps are merged by taking the maximum value per key. This ensures that a replica will always merge the highest (and thus latest) values from every replica, regardless if they are increments or decrements. To query the actual value of a particular replica, all the values in *posC* will be summed (i.e., all the increments of each replica), and then all the values from *negC* will be subtracted (i.e., all the decrements of each replica).

### 2.2.1.2 Operation-Based CRDTs

Operation-based CRDTs (or CmRDTs) handle synchronisation through the propagation of operations rather than states. Each replica applies local updates and broadcasts these operations to other replicas. The main principle behind operation-based CRDTs is that operations must be designed to be commutative so that they can be applied in any order leading to the same state [SPBZ11a, BAS14, BZP<sup>+</sup>12, Pre18].

In state-based CRDTs, operations can be idempotent; by design, merging the same state twice will not cause any issues. With operation-based

---

<sup>1</sup>The original specification from [SPBZ11a] utilises a vector instead of a map, as the authors assume that replica ID is a numeric value, starting at 0, and incremented for each new replica. We use a map instead and assume that the identifier can be any type of value.

CRDTs, this is not the case; applying an operation twice should lead to another state. This means that a system using operation-based CRDTs needs to ensure that all operations are delivered without duplication. This is typically achieved through a middleware providing reliable delivery<sup>2</sup>.

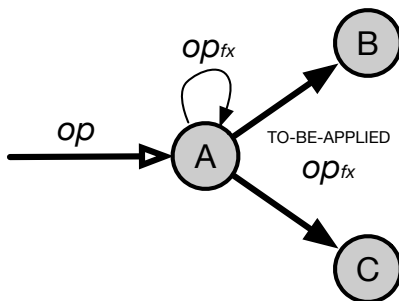


Figure 2.2: A representation of the operation-based CRDT replication process.

Figure 2.2 shows the typical replication process in a system using operation-based CRDTs. In the example, the system exists out of three replicas:  $A$ ,  $B$ , and  $C$ . An operation  $op$  is issued to replica  $A$ .

Before the operation is effectively applied, it can be transformed, e.g. mutated to include some metadata that may be necessary for ensuring commutativity. Many operation-based CRDTs will transform an operation by attaching a unique identifier or causal metadata to it (e.g. such as the Observed-Removed Set [SPBZ11a]). In operation-based CRDTs, we say the transformed operation  $op_{fx}$  is *generated*, applied locally (also known as *at source*), and propagated to the other replicas (also known as *downstream*).

**An Operation-Based Counter CRDT.** We now revisit the counter CRDT example using an operation-based approach. Algorithm 2 shows the specification of an operation-based counter. The design relies directly on the commutativity of the addition operation. The state only consists of the counter value itself, and updating it involves invoking the *inc* or *dec* operations. These operations will propagate (typically through broadcasting) the update operations to other replicas (as well as apply it locally).

<sup>2</sup>All operations from a particular replica need to arrive in the same order that they were sent, and with at-most-once (a.k.a. exactly-once) semantics [TvS06, BJ87]

---

**Algorithm 2:** Operation-Based Counter.

---

<b>state<sub>i</sub>:</b> $value_i: number$	
<b>query<sub>i</sub></b> $getValue(): number$	
$value_i$	
<b>update<sub>i</sub></b> $inc(n: number)$	
<b>broadcast</b> $inc(n)$	▷ At source generator
<b>update<sub>i</sub></b> $dec(n: number)$	
<b>broadcast</b> $inc(-n)$	▷ At source generator
<b>receive<sub>i</sub></b> $inc(n: number)$	
$value_i := value_i + n$	▷ Downstream effector

---

On reception of an operation, it will be applied to the replica state. Unlike the state-based approach, we do not need special logic for decrements and can implement them as negative increments.

### 2.2.1.3 Complex Operation-Based CRDTs

Designing new CRDTs that guarantee convergence is a complex task. Only for data types for which all operations commute, as is the case with a counter, can a CRDT be easily constructed since the resulting state will be equivalent regardless of the ordering in which operations are applied. As mentioned above, concurrency semantics are used to ensure that all concurrent operations commute [Pre18, RJKL11, BAS17]. These concurrency semantics are typically highly specific to particular CRDT designs. Many CRDTs (e.g. OR-Sets, MV-Registers, U-Sets, RGA, ...) implement their concurrency semantics through the use of unique identifiers and metadata [SPBZ11a, BZP<sup>+</sup>12, RJKL11, Pre18]. We will illustrate this approach by detailing the design of the Observed-Wins Set (OR-Set) CRDT [SPBZ11a], which ensures *add-wins* concurrency semantics. Add-wins semantics implies that concurrent *add* operations will always be applied after any *remove* operations.

**An Operation-Based Set CRDT.** The OR-Set achieves commutativity of operations by generating a unique identifier for every newly added item, making every add unique. From the user’s point of view, nothing changes: the client code interacts with the OR-Set using regular *add* and *remove* operations (which, as explained in Section 2.2.1, do not commute).

---

**Algorithm 3:** Operation-Based Observed-Removed Set, based on design from [SPBZ11a].

---

```
state: entries: Map(id → value), tombstones: Map(id → bool)
query getSet()
| {entries[i] |  $\forall i \in \text{keys}(\text{entries})$ }
update add(value)
| broadcast add(value, generateUniqueId())
update remove(value)
| broadcast remove(value, {i |  $\forall i : \text{entries}[i] = \text{value}$ })
receive add(value, id)
| if does not exist: tombstones[id] then
| | entries[id] := value;
| else
| | delete tombstones[id];
| end
receive remove(value, ids)
| for id in ids do
| | if exist: entries[id] then
| | | delete entries[id];
| | else
| | | tombstones[id] := true;
| | end
| end
```

---

However, at the implementation level, every time an element is added to the set, a (hidden) unique identifier is stored with it.

Algorithm 3 shows the specifications for the OR-Set. If an element is added twice, the set will logically only contain it once, but internally, the `add` update function will have generated two unique identifiers.

When an item is removed, replicas instruct others to remove only the items with the identifiers they have observed before. If the replica receiving the remove has not yet observed a certain identifier, it will keep track of the removed identifier as a *tombstone* and effectively delay the operation until after the add is received. This ensures that in the case of concurrent add and remove operations, the add will always be ordered before the remove, resulting in a commutative data type.

In a nutshell, the unique identifiers in OR-Sets have essentially two purposes: 1) encoding the *happened-before* relation of operations [Lam78]; e.g. remove operations that do not include a certain identifier must have happened before adds with that identifier, and 2) providing *add-wins* semantics for concurrent operations, i.e., concurrent adds will win over concurrent removes.

[BZP<sup>+</sup>12] proposed the Optimized OR-Set, which removes the need for tombstones by relying on a Reliable Causal Broadcasting (RCB) [BJ87] middleware. RCB ensures causal ordering for non-concurrent operations (along with reliable delivery), which makes it ideal for operation-based CRDTs. In the context of OR-Set CRDTs, causal delivery will always ensure that all `removes` come after their corresponding `adds`, eliminating the need for unique IDs and tombstones.

## 2.3 Pure Operation-Based CRDTs

Following the Optimized OR-Set [BZP<sup>+</sup>12], Baquero et al. [BAS14, BAS17] propose to reify and use the causality information from a Reliable Causal Broadcast [BJ87] (RCB) middleware. To this end, they introduced the Pure Operation-Based framework embodying such an implementation strategy. Their approach allows a structured way to encode concurrency semantics and ensure the causal ordering for non-concurrent operations (along with reliable delivery) [SPBZ11a, BAS14]. The framework employs a partially ordered log of operations (PO-Log) constructed with the causality information of the underlying RCB middleware. The state of

the data structure can be computed by observing this log and the log can be compacted to ensure that memory does not grow unboundedly.

### 2.3.1 Replication and Memory Management

Algorithm 4 describes the interaction between the RCB middleware and the pure operation-based CRDT framework. Each replica contains a particular state ( $s_i$  for replica  $i$ ), representing its PO-Log. The *operation(o)* method is called (e.g. by a CRDT implementation using the pure operation-based framework) when an operation  $o$  should be applied. It ensures that operations are broadcasted to other replicas and annotated with a logical timestamp on delivery ( $t$  in the algorithm description). It does this by invoking the *broadcast* method from the RCB layer. On delivery of these operations, the RCB layer will invoke the *deliver(t, o)* method from the pure operation-based framework after all causal dependencies operations have been received.

The framework introduces the concept of *causal redundancy* to keep the log compact. The idea is that a particular operation may make existing operations in the log redundant or that the arriving operation may be redundant itself. Two binary redundancy relations define rules determining this:  $R$  and  $R_{\_}$ .  $R_{\_}$  defines whether an arriving operation makes existing entries in the log redundant, and  $R$  defines if a newly arriving operation should be stored in the log. The concrete CRDT implementation needs to provide the definitions for these relations, which will be checked upon delivery as shown in Algorithm 4.

The framework can also determine when operations are *causally stable*, i.e., they have been observed on all replicas, and trim causal information for their log entries. Baquero et al. [BAS17] define causal stability as follows:

**Definition 2.3.1 (Causal Stability)** *A timestamp  $\tau$ , and a corresponding message, is causally stable at node  $i$  when all messages subsequently delivered at  $i$  will have timestamp  $t > \tau$ .*

This implies that new operations can never be concurrent with causally stable operations, and as such, their causal metadata (such as timestamps) is no longer needed. The RCB layer can determine causal stability by comparing the vector clocks of incoming messages and decide whether a

particular timestamp has been observed by all nodes. Whenever a particular timestamp is causally stable, the `stable` function will be invoked by the RCB layer, and the framework will compact stable operations that are returned by the `stabilize` function. It does this by replacing (removing) the associated timestamp with the bottom (null) element. Similarly to the redundancy relations, the `stabilize` function has to be provided by any CRDT implementation built on the framework.

---

**Algorithm 4:** (Simplified) distributed algorithm for a replica  $i$  showing the interaction between the RCB middleware and the pure op-based CRDT framework. Based on design from [BAS17].

---

```

state:  $s_i := \emptyset$ 
on  $operation_i(o)$  :
    | broadcast $_i(o)$ 
on  $deliver_i(t, o)$  :
    |  $s_i := (s_i \setminus \{(t', o') \mid (t', o') \in$ 
    |    $s_i \cdot (t', o') \mathbf{R}_\perp(t, o)\}) \cup \{(t, o) \mid (t, o) \mathbf{R} s_i\}$ 
on  $stable_i(t)$  :
    | stabilize $_i(t, s_i)[(\perp, o)/(t, o)]$ 
    
```

---

### 2.3.2 A Pure-Operation Based Set

We now show a pure operation-based set, which, similar to the OR-Set, implements add-wins concurrency semantics. Table 5.1 details the implementation and is grouped as follows: (1) functions that are used by the

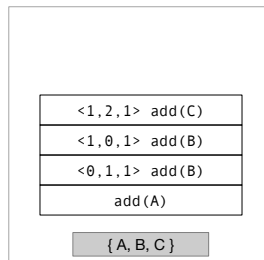


Figure 2.3: The internal state of an AW-Set. One operation is causally stable and, as such, does not contain a timestamp. Together, the operations form the state  $\{A, B, C\}$ .

framework and that dictate the interaction between new operations and entries in the log, and (2) procedures that can be invoked by the user for state serialisation or mutations.

The  $R$  relation for the add-wins set defines that the `clear` and `remove` operations will never be stored in the log.  $R_{\_}$ , on the other hand, defines that an arriving operation  $o$  will make any stored operations (in the log) redundant if and only if the stored operation  $o'$  causally happened before the arriving operation (i.e.  $t' < t$ ) and the arriving operation is acting on the same set element, or the arriving operation is a `clear` (i.e., which removes all happened-before elements). For example, a `remove(X)` will make a previous `add(X)` redundant, and a `clear` operation will remove all previous log entries. The combination of both rules ensures that `add` operations will always 'win' from concurrent operations. The implementation of `stabilize` defines that all causally stable operations will be stripped from their timestamps (to preserve memory consumption). Additionally, the log will only contain distinct `add` operations at any point in time. To query the state, a query function can extract each added element from the log (as shown in the `toList` function) and serialise it into an actual set data structure.

Figure 2.3 shows how the PO-Log of an Add-Wins (AW-Set) set replica (in a system of three replicas) might look. It contains four `add` operations, which form the state  $\{A, B, C\}$ , depicted in grey. Three of these operations include causality information from the underlying RCB middleware, i.e. they carry a vector clock. The final `add(A)` operation has been stripped from causality information, as the operation is causally stable.

Table 2.1: Semantics for the add-wins pure-op set, based on the approach in [BAS17].

Pure	$(t, o) \mathbf{R} s$	$= \text{op}(o) = (\text{clear} \vee \text{remove})$
	$(t', o') \mathbf{R}_{\_} (t, o)$	$= t' < t \wedge (\text{op}(o) = \text{clear} \vee \text{arg}(o) = \text{arg}(o'))$
	$\text{stabilize}(t, s)$	$= s$
User	$\text{toSet}(s)$	$= \{v \mid (\_, [\text{op}=\text{add}, \text{arg}=v]) \in s\}$
	$\text{add}(e)$	$= \text{operation}([\text{op}=\text{add}, \text{arg}=e])$
	$\text{remove}(e)$	$= \text{operation}([\text{op}=\text{remove}, \text{arg}=e])$
	$\text{clear}()$	$= \text{operation}([\text{op}=\text{clear}])$

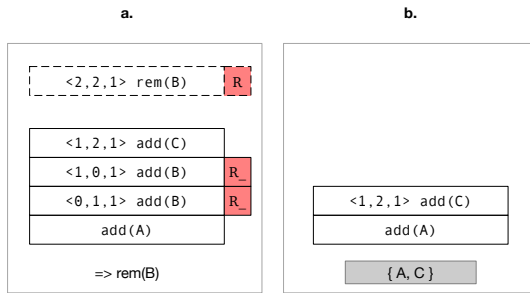


Figure 2.4: The internal states of an AW-Set, after receiving a `remove` (`rem`) operation, and after the operation has been applied.

Figure 2.4 illustrates the internal state and the PO-Log of the AW-Set depicted in Figure 2.3 after receiving a `remove`(`B`) operation (depicted in the *a.* box) and after the operation has been applied (depicted in the *b.* box). Initially, the log consists of an operation which is causally stable (the `add(a)`), and three other operations which are not yet stable. Looking at the vector clocks, we can observe that the log has two concurrent operations, both of which add element `B`. When the arriving `remove(B)` is checked against these stored operations, both previous `add(B)` operations will be marked as redundant by the  $R_{\_}$  relation (as the operations have the same key, and are causal predecessors). Additionally, the arriving operation itself is immediately marked as redundant by the  $R$  relation of the AW-Set semantics (all `remove` and `clear` operations are immediately redundant) and as such, it will not be added to the log. The box denoted by *b.* shows the final result of applying `remove(B)`: no entries for adding element `B` remain, and the removal operation itself was not added to the log. Thus, the replica state becomes  $\{A, C\}$ .

**Remove-Wins Semantics.** Table 2.2 shows the Remove-Wins Set (RW-Set) CRDT specification in the pure operation-based CRDT framework [BAS17]. It provides remove-wins concurrency semantics, meaning that `add` operations will always be ordered before concurrent `remove` operations.

Unlike with the AW-Set, the RW-Set stores `remove` operations in the PO-Log, as concurrent `add` operations that should be made redundant by the `remove` can arrive at any point in time. This is exactly what is defined by the  $R$  relation: any arriving `add` operation will be immediately

Table 2.2: Semantics for the remove-wins pure-op set, based on approaches from [BGB20b, BAS17].

Framework	$(t, o) \mathbf{R} s = \text{op}(o)=\text{add} \wedge \exists(t', [\text{op}=\text{remove}, \text{arg}=\text{arg}(o)]) \in s \cdot t \sim t'$ $(t', o') \mathbf{R}_- (t, o) = (t' < t \wedge (\text{op}(t') = \text{add} \vee \text{arg}(o) = \text{arg}(o')) \vee ((t' < t \vee t \sim t') \wedge \text{op}(o) = \text{remove} \wedge \text{op}(o') = \text{add} \wedge \text{arg}(o) = \text{arg}(o'))$ $\text{stabilize}_i(t, s) = \{(t', o) \mid \forall(t', o) \in s \cdot t \neq t'\} \cup \{\forall(\perp, [\text{op}=\text{add}, \text{arg}=e]) \mid (t', [\text{op}=\text{add}, \text{arg}=e]) \in s \cdot t = t'\}$
User	$\text{toSet}(s) = \{v \mid (\_, [\text{op}=\text{add}, \text{arg}=v]) \in s\}$ $\text{add}(e) = \text{operation}([\text{op}=\text{add}, \text{arg}=e])$ $\text{remove}(e) = \text{operation}([\text{op}=\text{remove}, \text{arg}=e])$ $\text{clear}() = \text{operation}([\text{op}=\text{clear}])$

redundant if there is a **remove** in the log for the same element. The  $R_-$  relation follows this by defining that any previous **add** operation, stored in the log, will become redundant if a following or concurrent **remove** operation arrives for the same element. We also remove any **add** operations if a causally following **add** on the same element arrives to avoid needless duplication of add records. To keep the PO-Log compact, the RW-Set 1) removes all **remove** operations from the log when they become causally stable, and 2) removes all timestamps from causally stable **add** operations.

## 2.4 State of the Art of CRDT Implementations

In this section, we discuss the state-of-the-art CRDT implementations and designs to put our contributions in context.

### 2.4.1 Approaches to Replication and Distribution

As mentioned in Chapter 1, much work focuses on designs and specifications for CRDTs. Initially, many implementations focused on geo-distribution, but with the advent of local-first applications, we have witnessed a shift in implementation approaches. This section explores various

implementations and their distributed design and ends with an overview of the open problems that motivate our work.

**Local-First.** Among local-first software approaches, the most popular CRDT implementations are Automerge [Aut] and Yjs [Yjs]. Automerge is based on an implementation of JSON CRDTs [KB17] and targets web-based platforms where copies of data are stored directly on end-user systems. It implements a system where JSON-like documents can be mutated and safely shared without any possibility of conflicts. However, Automerge does not include a built-in distribution layer; system designers are responsible for implementing the replication mechanism. This allows the framework to be used in various settings over various connectivity mediums.

As a concrete example, consider dynamic networks. Automerge enables new nodes to construct their initial state by receiving a serialised copy of the state of one of the other replicas. Subsequent updates can then be shared with the new node. However, this approach burdens system developers to manage concurrent updates during the node initialisation in an ad-hoc way.

Yjs is a similar framework targeting web-based applications that enable generic shared documents. It has a distribution layer, based on the YATA CRDT framework [NJDK16], that can replicate data over WebRTC and other protocols. The framework can easily be adapted to work with different network protocols. While YATA does support dynamic networks, the approach does not scale with large networks. When a node (re-)joins the network, online replicas will attempt to send any missing updates. In the case of a new node, this would be the entire state of the replica. As every replica will apply this procedure, the network may be flooded with duplicate operations and states.

**Geo-Replication.** When looking at geo-replicated systems, AntidoteDB [ATB<sup>+</sup>16, Ant] and, more recently, Azure CosmosDB [GPGP18, Cos] offer industry-strength solutions. AntidoteDB [ATB<sup>+</sup>16] is a comprehensive geo-distributed key-value store that uses CRDTs to ensure availability, low latency, and partition tolerance. AntidoteDB offers various CRDT types for developers to use as data values. It employs Cure, a distributed storage system implemented in Erlang, to guarantee causal consistency

and atomicity. Erlang’s message-passing system enables modular replica discovery and operation/state propagation. Although AntidoteDB supports dynamic network reconfiguration [LSB<sup>+</sup>19], detailed descriptions of the used approach are not found in published literature.

Azure Cosmos DB is a NoSQL database that uses CRDTs as part of its core type system. While it lacks clear publications on its internal functioning, documentation highlights that it mostly relies on Last-Write-Wins (LWW) mechanisms in combination with custom conflict resolution policies, more akin to cloud types [BFLW12].

**Language-Centric Approaches.** Lasp[MVR15], like AntidoteDB, relies on Erlang for its distribution capabilities. It provides a general-purpose CRDT framework that is directly accessible from Erlang to propagate state changes to connected nodes through Riak DT. While nodes can dynamically join, state-based replicas can only obtain a full state if operations have been applied before. Operation-based CRDTs will be missing any operations applied before the join, and the system developer will need to implement manual techniques to ensure proper synchronisation.

Other approaches, such as ConSysT [EKMS19], provide programming language abstractions for specifying consistent levels. They allow developers to safely mix and define their system’s requirements. The result is a mixed consistency approach that not only relies on CRDTs but can also adapt to different synchronisation models.

Triumvirate [Myt19] provides a DSL tailored towards the implementations of distributed rich internet applications. It supports a suite of RDT to represent distributed state and additionally allows for the distribution of programming logic across servers and clients.

**Overview:** All in all, the overviewed approaches and most other designs [RJKL11, WUM10, Klo10] generally take an ad-hoc approach to replication and distribution aspects. This may make it harder to reuse and extend such systems. Earlier work on Group Communication Systems (GCS) [Gol92, ADKM92] notes the importance of supporting dynamic environments in replicated environments and proposed mechanisms to this end. However, we observe that the concepts introduced by GCS have not been adapted for CRDT-based systems; there are no systematic approaches for the handling of dynamic networks. Additionally, as mentioned in the in-

roduction, the hard requirement for reliable causal broadcasting by the majority of frameworks, middleware, and libraries for CRDTs hampers the reactivity of such systems [BFG<sup>+</sup>12].

## 2.4.2 Memory Management in CRDTs

While CRDT designs do not rely on synchronisation to impose an ordering between operations in the system, many CRDT designs track the causality of operations. As explained previously in Section 2.2.1.3, this is often done by tagging operations with some metadata, either through direct logical clocks or values that represent the causal origin of the operations in some way. For example, OR-Set CRDTs (Section 3.3.2) tag values with (hidden) unique identifiers and tombstones. This ensures that basic causality can be tracked between operations, as concurrent operations will not contain the unique identifiers of operations that they have not observed.

Logging such metadata can be problematic as it may grow unboundedly if not removed. Garbage collecting useless data from eventually consistent (EC) systems is not trivial, as different system parts may be in different states. One of the major problems of garbage collection in EC systems is the lack of fixed synchronisation between replicas that can be used to determine when data becomes garbage.

Specific approaches have been proposed to address this issue directly in the CRDT designs [BZP<sup>+</sup>12, RJKL11]. For example, Lasp [MVR15] uses optimised OR-Sets, though some metadata removal mechanisms are disabled when replicas are offline. While effective, these techniques are mostly specific to certain CRDT designs and cannot be trivially applied to other designs.

The Pure Operation-Based CRDT framework [BAS17], on the other hand, provides general-purpose techniques through the reification of causal clocks in the form of a partially ordered log. This introduces a systematic approach to the handling of memory management. Sadly, mainstream frameworks have yet to adopt pure operation-based CRDTs and, as such, do not benefit from this approach.

Yjs [NJDK16] implements a time-based garbage collection scheme. This approach imposes a bound on when concurrent operations can be applied, which helps deduce which causal metadata can be safely removed. However, this approach has limitations in offline settings or when the system experiences significant latencies. The original YATA paper [NJDK16]

discusses an alternative approach that closely aligns with the techniques introduced by the Pure Operation-Based CRDT framework. However, to the best of our knowledge, this alternative approach has not been integrated into Yjs due to concerns regarding the potential network overhead.

Automerge makes use of a log to keep track of all applied operations. This approach can be practical for keeping a complete history of changes applied to a document. Although it preserves a full historical record, the continuous accumulation of changes may lead to excessive memory usage, which requires careful consideration by system developers.

**Overview:** While various techniques have been developed to manage memory in CRDTs, each comes with its trade-offs and limitations. Effective memory management remains a critical aspect for the implementation of CRDT systems to ensure both efficiency and scalability. The Pure Operation-Based CRDT framework introduces a systematic way to reason about the causality of operations and, as a result, simplifies CRDT design. It provides an important inspiration for our work, where we set forth to provide systematic solutions to most of the described problems.

### 2.4.3 Composition and Nesting in CRDTs

Most research in replicated data types has focused on creating a portfolio of conflict-free data structures such as counters, sets, and linked lists [SPBZ11a, WUM10, RJKL11, BZP<sup>+</sup>12, Sha17, Pre18]. However, the composition and nesting of those CRDTs have received less attention.

The composition of predefined replicated structures is possible in a few frameworks like Automerge [KB17], AntidoteDB and Lasp [MVR15]. While Automerge and AntidoteDB allow arbitrary nesting of lists and maps, they offer limited flexibility in merging semantics, as custom CRDTs cannot be used. Lasp supports functional transformations over existing CRDTs in the language, enabling some level of composition. The approach is dataflow-centric, meaning that structure must be imposed through functional operations and destructors. This process is complex and error-prone if large and complex structures must be created.

Recently, novel approaches have arrived for the implementation of implementation of Composed CRDTs [WMM20]. They provide a systematic approach to composition on top of the pure-operation CRDT framework, though only allow static structures where the structure cannot be modi-

fied at runtime. Currently, this approach is being extended [WQK<sup>+</sup>23] to support the general nesting of pure operation-based CRDTs. This work was developed concurrently with our research and offers many similarities to the nesting framework which we will present in this dissertation.

**Security and Authorisation.** Flexible composition techniques are not only essential for managing data structures but also for tracking metadata related to security and authorisation in a system. Previous work on eventually consistent (EC) data stores has highlighted the challenges of applying general authentication and security techniques to weakly consistent systems [RB94, LC18]. One significant challenge is preventing data tampering while maintaining availability, an unresolved issue particularly relevant to programming languages.

Ensuring trust in EC systems is complex. The ACGreGate framework by Weber et al. [WB18] addresses this by proposing a CRDT structure designed to store and propagate authorisation policies within a weakly consistent system. This approach ensures that authorisation metadata is correctly replicated and enforced across the system, maintaining both security and availability.

While CRDT frameworks like Automerge and Lasp offer some support for composition and nesting, there remains a significant gap in addressing security and authorisation within these systems. Ensuring that authorisation metadata is consistently and correctly replicated is crucial for the trustworthiness of EC systems, highlighting the need for further research and development in this area.

**Overview:** Mainstream libraries and frameworks allow developers to use nested and composed CRDTs but are limited to particular sets of CRDTs, and developers cannot configure their own concurrency semantics for nested structures. Only recent work has investigated systematic, non-ad-hoc approaches to nesting and composition, which we believe are crucial to enable more widespread usage of CRDTs and tackle implementation concerns such as security.

## 2.5 Conclusion

In this chapter, we reviewed the state-of-the-art in CRDT programming and the required background for this dissertation. We explored various approaches to CRDT design and highlighted the impact of different design choices on the guarantees for replicated systems.

Many state-of-the-art implementations exist that address specific programming concerns we highlighted. However, most of these solutions rely on an ad-hoc approach or are limited to specific CRDT framework designs. Continued research is essential to develop robust, scalable, and, most importantly, flexible and systematic CRDT designs and frameworks.

We identify that the Pure Operation-CRDT framework provides a concise approach to developing and implementing CRDTs. In this framework, replication and memory concerns are abstracted away from the concrete semantics of data structures. This allows for clean and clear implementations, minimising accidental complexity and the changes required to correct mistakes in CRDT designs. In the following chapters, we will build on the ideas of the Pure Operation-CRDT framework to provide systematic solutions to the concerns that we expressed in our problem statement, namely that many designs use ad-hoc approaches to replication, memory management, and composition. The next chapter will first detail Flec, our implementation framework, which is used as a laboratory for CRDT experimentation.



## Chapter 3

# Flec: a Programming Framework for Eventually Consistent Systems

CRDT research has primarily focused on providing formal specifications of different data types (e.g. OR-Sets, replicated growable arrays, embeddable counters and more) [SPBZ11a, RJKL11, BGYZ14, ZBPH14, BAL16], but fewer work has focused on embedding CRDT in actual language implementations [MVR15, KB17]. Developers using existing libraries need to handle many distribution aspects themselves, such as how to discover new network acquaintances and how they will cope with a dynamically changing system [BGB19]. This greatly raises the barrier to utilising CRDTs.

We built Flec, a versatile programming framework for eventually consistent systems, as a laboratory for experimenting with CRDTs. Flec offers a modular and extensive approach for developing and using CRDTs more easily. Key features of Flec include:

- A flexible networking framework that is adaptable to different networking protocols, simplifying the integration of CRDTs across various environments.
- A Meta Object Protocol (MOP [KdRB91]) provides an API exposing the inner workings of CRDTs, facilitating the creation of new CRDT variants by allowing developers to intercept and adapt the replication process at relevant points. For example, a CRDT im-

plementor can use this API to intercept operation propagation and modify their behaviour.

- From a technical viewpoint, our implementation is TypeScript-based, a modern language chosen for its portability and wide platform support. This makes Flec an effective tool for experimenting with and creating new replicated data structures in real-world applications.

Overall, Flec aims to lower the barrier to utilising CRDTs in application development, providing a comprehensive and adaptable framework for researchers and developers. We use it as the foundation for the work described in this dissertation, with all implementations working on top of Flec.

### 3.1 The Flec Programming Framework

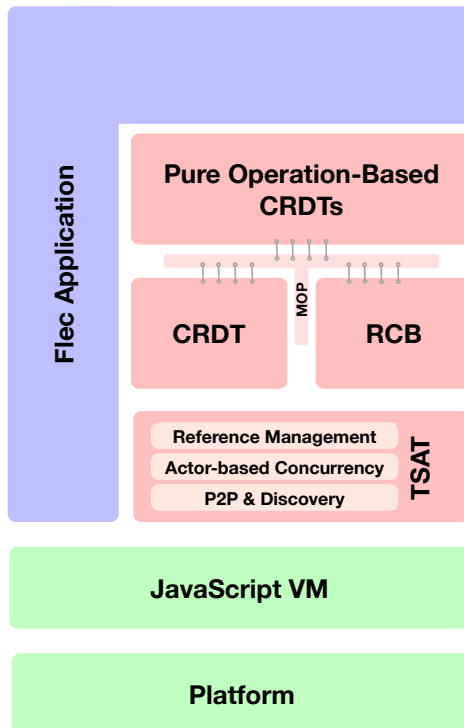


Figure 3.1: Flec architectural overview.

Figure 3.1 gives a general overview of the Flec Programming Framework. Flec and its components are implemented in TypeScript, a typed superset of JavaScript. This allows the framework to run on various platforms that support a JS-runtime, such as Web Browsers and IoT devices.

The core of Flec is a component named TSAT, which is a layer that incorporates concepts of ambient-oriented programming for TypeScript [CMGB<sup>+</sup>07, DVCM<sup>+</sup>06] for distributed programming with objects. In ambient-oriented programming, developers have an actor-based programming model where actors can communicate and coordinate over dynamic networks.

Flec is engineered to be network agnostic and allows communication pipelines to be implemented using any underlying transport mechanism. Depending on what platform Flec is running on, the exact means of transportation can be different: web browsers could be using WebSockets or WebRTC, on embedded devices (such as the ESP32, a lightweight, power-efficient integrated system-on-a-chip platform) it could be over pure TCP sockets.

Other significant components are the Reliable Causal Broadcasting (RCB) layer for ensuring causal and reliable delivery guarantees, the CRDT layer for generic CRDT implementations, and the Pure Operation-Based CRDT framework. They are tightly coupled together through the MOP, which allows flexible modifications to the replication system, and rely on the TSAT layer for distribution capabilities. These components will be described in detail in later sections of this chapter.

## 3.2 Ambient-Oriented Programming Programming in TSAT

As mentioned, TSAT follows the ambient-oriented programming (AMOP) paradigm [CMGB<sup>+</sup>07] that allows for the simplified development of distributed applications. Following the principles of this paradigm, TSAT features an actor-like message-passing system that can work over a network in a distributed setting. This section describes how TSAT provides concurrent and distributed programming constructs through non-blocking message passing between actors and the synchronisation primitives that can be used to coordinate asynchronous operations.

### 3.2.1 Concurrent Programming

TypeScript is a typed superset of JavaScript, a prototype-based object-oriented programming language. JavaScript runtimes use an event-loop execution model, which processes events and manages the queueing of tasks and their eventual execution. Events in this model are typically handled asynchronously, through callbacks, promises, or `async/await`.

Flec/TSAT extends this with support for actor-based concurrency, where objects are owned by actors. Within a single-actor environment, objects can be referenced by other objects through *direct references*, as is the default in TypeScript. Access to objects residing in other actors is done via *far references* [MMF01]. While communication via direct references is synchronous, communication via far references is always asynchronous. In this section, we explain the concurrent programming model of TSAT through a *PingPong* application.

#### 3.2.1.1 Actors in TSAT

Actors in TSAT define a single thread of execution and act as a container for objects. Actors communicate with other actors through asynchronous message passing. Each actor has a mailbox (a message queue), and an internal processing thread atomically processes messages from the mailbox. Actors define a *behaviour* object that specifies their interfaces, i.e. the set of messages that an actor understands. When creating an actor, a far reference to the behaviour is returned so that other actors can send messages to it.

Listing 3.1 shows how a ping actor can be created with TSAT by invoking the method `newActorWithBehaviour` on a TSAT context. The method's first argument dictates the actor's name; the second argument should be the type class used to construct the behaviour object, which in TSAT extends the `ActorBehaviour` class. In the example, we create the `ActorABehaviour` class. Providing an anonymous (unnamed class) as behaviour is also possible. If an actor behaviour defines an `init` method, this method will be invoked on the successful initialisation of the actor. In our case, `init` prints a hello world message on initialisation. The behaviour also defines a `ping` method that prints a message and returns a string `pong`. The `newActorWithBehaviour` method returns a far reference to the behaviour of this class, which is an instance of the `Far-`

Listing 3.1: Code snippet of a 'Ping Pong' application showing actor creation in TSAT.

```
1 const ctx = new TSAT("vma");
2
3 let actorA = ctx.newActorWithBehaviour("Actor A", class
4   ActorABehaviour extends ActorBehaviour {
5   init() {
6     console.log("[A] Hello, World!");
7   }
8   ping() : string {
9     console.log("[A] Received a Ping, returning a pong");
10    return "pong";
11  }
12 });
```

`Ref<ActorABehaviour>` class. Other actors can use the far reference to send asynchronous messages to the actor. It is possible to supply extra arguments to the `newActorWithBehaviour` method; these will be passed (by copy) to the `init` function as arguments.

### 3.2.1.2 Asynchronous computations

We will now elaborate on the message-passing semantics of far references by extending the previous example.

Listing 3.2: Code snippet of a 'Ping Pong' application that shows asynchronous message sends.

```
1 ctx.newActorWithBehaviour("Actor B", class extends ActorBehaviour
2   {
3   async init(actorA) {
4     console.log("[B] Sending a ping");
5     let result = await actorA.ping();
6     console.log("[B] Received a response:", result)
7   }, actorA);
```

In Listing 3.2 we show the creation of a new actor, Actor B. On initialisation, this actor uses the far reference of Actor A to send a `ping` message. Sending a message to a far reference returns immediately, never blocking the actor. However, there are scenarios where it is necessary to collect the result of asynchronous messages. This is accomplished using promises, which are objects that act as placeholders for the future results of asynchronous computations [YBS86].

TSAT integrates these asynchronous computations with an extended version of JavaScript promise objects. Consequently, all message invocations return such promises.

In our example, the `ping` method returns a promise. Actor B will *await* this promise to get the computation result and subsequently print it. The `await` operator, part of the JavaScript standard, allows function execution to pause until the promise is resolved.

In TypeScript, a promise is a fundamental concept for handling asynchronous operations. It represents a value that may not yet be available but will be resolved in the future. A promise can be in one of three states: pending, resolved, or rejected. While pending, the operation is ongoing. Once completed, the promise is either resolved with a value (if successful) or rejected with a reason (if failed).

Promises in TypeScript are strongly typed, allowing developers to specify the type of value that a Promise will resolve with. For example, a Promise that resolves with a number can be typed as `Promise<number>`. Promises can be consumed using the `.then()` method, which registers callbacks to receive the Promise's eventual value. Additionally, the `.catch()` method can be used to handle errors.

### 3.2.2 Chainable Promises in TypeScript and TSAT

TSAT extends on TypeScript promises by allowing them to be chained with the `ChainablePromise<T>` class type. Any operation on a far reference will always return a chainable promise. Chainable promises allow operations to be directly applied to promises without needing to wait for them to be resolved. The operation will be buffered and applied to the promise once resolved. Applying an operation on a promise will return a new chainable promise that will be resolved once the buffered operation is applied and resolved.

Listing 3.3: Code snippet demonstrating promise chaining.

```
1 let res : ChainablePromise<string> = ref.ping("ABC");  
2 res.charAt(0).then(char => {  
3     console.log(`The first char of the string equals ${char}`);  
4 });
```

Listing 3.3 shows an example code snippet that sends a ping to a far reference. The asynchronous result of the operation is a string, which means that in TSAT the operation will return a `ChainablePromise<string>`. As explained, it is possible to immediately execute operations on the promise, as is done in the example by calling `charAt(0)` on the promise. This operation will be buffered until the ping is resolved, after which it will be applied to the result. Finally, the `charAt` operation itself will return a new `ChainablePromise<string>`, allowing for successive chaining of operations. In our case, we attach a resolution callback on the promise that logs the result with `then` method.

### 3.2.2.1 Async/Await

JavaScript and TypeScript provide support to simplify working with promises by means of `async/await` that provides a more convenient and readable way to handle asynchronous operations. An `async` function in TypeScript is a function that implicitly returns a promise. Inside an `async` function, you can use the `await` keyword to pause the execution of the `async` function until a promise is resolved. If the promise is resolved, the function continues with the resolved value. If the promise is rejected, an error is thrown, which can be caught using `try-catch` blocks within the `async` function.

Instead of chaining multiple `.then()` and `.catch()` methods, a developer can write sequential code. Instead of nesting callbacks, you can `await` multiple promises in a series, executing each statement after the previous asynchronous operation. It is important to note that the event loop is not blocked when an `async` function is paused. `Async` functions are implemented through `co-routines`, where control will be passed to other parts of the system.

Listing 3.4: Code snippet demonstrating promise chaining combined with an `await` statement.

```
1 let char = await res.charAt(0);  
2 console.log(`The first char of the string equals ${char}`);
```

TSAT allows chainable promises to be used with the standard `async/await` syntax. Listing 3.4 shows the previous code example, modified to use `await` instead of an explicit `then` operation with callback. In this case, the log will only be called once the promise is resolved. This simplifies the code and improves its readability.

### 3.2.2.2 Parameter Passing Semantics

When sending a message to a far reference, arguments are either passed by copy or passed by reference, depending on the type. All TypeScript primitives (e.g., strings, numbers, booleans, ...) are always passed by copy. Object types can be passed by copy if they implement the *Isolate* type, which forces the types to be serialisable. All other objects will be passed by reference, where the method that handles the message will obtain a far reference to the passed object. The TSAT type system ensures that it is impossible to send a (non-isolate) object to a function that does not expect a far reference.

### 3.2.3 Distributed Programming

Besides being the unit of concurrency in Flec/TSAT, actors are also the unit of distribution. Actors can communicate with other actors from other networked devices through asynchronous messages in the same way they would with local actors. Figure 3.2 conceptually shows how objects can be referenced by far references. Far references can cross virtual machine (VM) and device boundaries. In this section, we describe how actors can discover other actors hosted on other devices in a network, i.e., TSAT service discovery, and how the programming model deals with partial failures of devices.

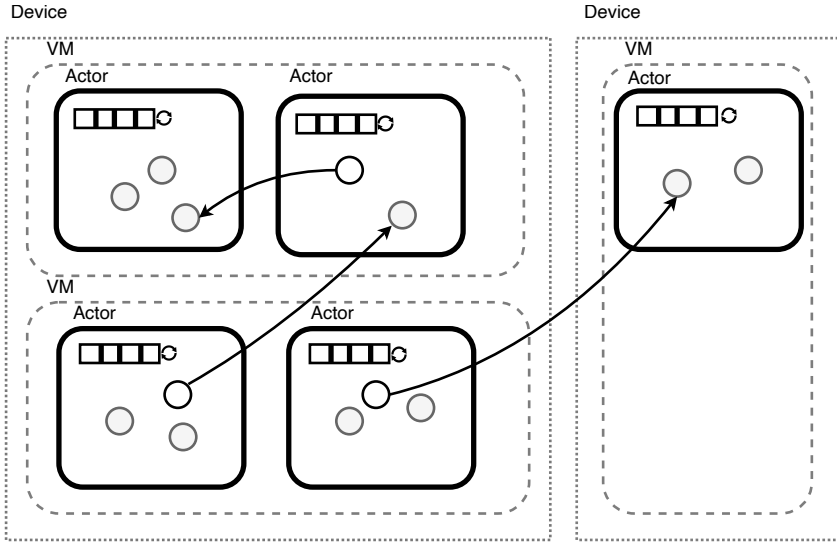


Figure 3.2: Far reference can point to objects on local or remote actors.

### 3.2.3.1 Service discovery

TSAT implements ambient acquaintance management [DVCM<sup>+</sup>06] from AMOP and incorporates constructs to export (i.e. publish) objects in the network and to subscribe to exported objects from different actors (running on the same or different machines). Exporting uses the `export` method on an actor or actor behaviour object. Objects are exported with a particular *tag name*. Tag names are strings used by actors to identify and discover objects and obtain far references to the discovered objects. Exported objects can be discovered using the `discover` method on an actor or actor behaviour object. The method takes the tag name of objects that need to be discovered, their expected type, and a callback. TSAT ensures that only objects with the specified type can be discovered to enforce the type safety of all operations.

In Listing 3.5, we revisit the ping-pong application, except that actors are automatically discovered in the network. Particularly, we instantiate a new actor with initialises and export a new `PingPong` object. Instead of defining the ping-pong behaviour in the actor itself, we instead separate its logic to a `PingPong` class. The `PingPong` class, defined in Listing 3.7, implements two methods: `ping` and `getCounter`. `ping` accepts a string input and will log the operation, increment a counter, and return the

Listing 3.5: Code snippet showing a TSAT actor exporting a PingPong object on the network.

```
1 ctx.newActor("Actor A", (actor) => {
2     console.log("[A] Exporting PingPong");
3     actor.export(new PingPong(), "PingPong");
4 });
```

Listing 3.6: Code snippet showing a TSAT actor discovering a PingPong object and sending a message to its far reference.

```
1 ctx.newActor("Actor B", (actor) => {
2
3     actor.discover(PingPong, "PingPong", async (ref: FarRef<
4     PingPong>) => {
5         console.log("[B] Discovered PingPong, sending a ping");
6
7         let reply = await ref.ping("B");
8         console.log(`[B] Got reponse: ${reply}`)
9
10        let logger = new Logger("[B] [Logger]");
11        let val = await ref.getCounter(logger);
12        console.log(`[B] Got counter value: ${val}`)
13    });
14 });
```

Listing 3.7: PingPong Class.

```
1 class PingPong {
2     counter: number = 0
3
4     ping(name: string) : string {
5         console.log(`[P] Got ping from ${name}`);
6         this.counter++;
7
8         return "pong";
9     }
10
11     getCounter() : number {
12         return this.counter;
13     }
14 }
```

string "pong". `getCounter` will return the value of the counter. The newly instantiated object is exported with the "PingPong" tag.

In Listing 3.6, we show how another actor is discovering published objects with the "PingPong" tag. A discovery callback can be registered with the `discover` method on an actor or actor behaviour object. Every time a new object on the network is discovered with the "PingPong" tag and the `PingPong` type, the callback will be called with a far reference to the discovered object. The callback sends `ping` messages to any discovered object and waits for its response. Once it has received, it will send a `getCounter` message to obtain a count of the number of ping messages sent.

### 3.2.3.2 Delivery Guarantees

Far references are designed to remain functional even when a system has partial failures, e.g., when certain system nodes might be disconnected. When TSAT attempts to deliver a message but detects that the recipient node is offline, it notifies all far references pointing to the node to buffer all messages. TSAT will regularly check the connection state and attempt to restore the connection. If this is successful, it will notify all far references to release their buffered messages.

TSAT ensures at-most-once delivery guarantees (also known as exactly-once delivery) [TvS06]. As such, messages are guaranteed to be delivered (in a correctly functioning system, when there are no partial failures) without duplication.

### 3.2.4 Networking and Communication Channels

TSAT will establish network communications with other TSAT instances that it discovers over the network. This allows applications to communicate with other actors running on remote machines. TSAT introduces the concept of channels, where different communication mechanisms can be used to link instances. Section 3.2.4 lists the available channel types for TSAT.

By default, Flec uses a P2P approach over TCP/UDP for network discovery, similar to the AMOP AmbientTalk language. This means it broadcasts instance information (which includes metadata of exported objects) using UDP multicasting/broadcasting and listens for such broadcasted information from other peers in the network. If a TSAT instance receives broadcasted information from other peers, it will register the instance in its remote instance table. It then checks the remote instance's metadata to see if any of the exported objects matches a tag (and type) that any of the local actors is trying to discover. If so, it will establish a TCP connection with the remote instance and construct a far reference for the newly discovered object. This reference will then be passed as an argument to the discovery callbacks for all the actors who are subscribed to the associated tag

TSAT features a channel for MQTT [HTSC08] as this network transport layer is popular in IoT systems. Listing 3.8 shows how TSAT can be configured to use this channel type. First, an instance of the MQTT channel type is instantiated, with the broker URI set to `mqtt://mqtt-broker.local:1883`. The channel will maintain a connection with the specified broker and use it for peer discovery and message sending. It is registered in Flec by using the `addChannel` method on the TSAT context object.

Using a different channel type does not require changes to the application code. All TSAT functionality will remain the same, regardless of the transport system used. It is even possible to register several channels simultaneously, have TSAT communicate and discover peers concur-

Table 3.1: Channel Types.

<b>TSAT: Supported Channel Types</b>	
<code>P2PChannel</code>	Provides a TSAT channel that communicates over TCP and UDP. Peer discovery happens through UDP multicasts/broadcasts, P2P message sending through TCP to discovered peers.
<code>MQTTChannel</code>	Provides a communication channel over MQTT. Allows TSAT to connect to an MQTT broker for discovery and message sending.
<code>WebChannel</code> (prototype)	Initial version of a channel that uses WebSockets with a certain server for peer discovery, and WebRTC for direct P2P message sending.

Listing 3.8: Code snippet showing MQTT configuration in TSAT

```

1 import { MQTTChannel } from "./MQTTChannel";
2 const mqtt = new MQTTChannel("mqtt://mqtt-broker.local:1883");
3 tsatCtx.addChannel(mqtt);

```

rently through different communication mechanisms. These details are abstracted away for application developers; their code only needs to reason about exporting and discovering objects with tags.

TSAT has initial support for a channel which uses WebSockets and WebRTC as transport layers. These technologies will allow Flec to be usable in the context of web browser-based applications.

### 3.2.4.1 Custom Channels

Following the philosophy of 'Flec as a laboratory,' we provide an API for implementing and experimenting with different communications technologies. The API allows for the implementation of new channels that use custom communication mechanisms or transport layers. To achieve this, a custom class has to be implemented (extending the `Channel` class) that provides primitives for 1) direct message sending, 2) message broadcasting, and 3) network event handling.

Table 3.2: Core Channel Interface.

<b>TSAT: Core Channel Interface</b>	
<b>Message Sending Methods</b>	
<code>sendMessage(msg: Msg, dest: FarReference)</code>	Invoked when TSAT wants to send a message <code>msg</code> to destination <code>dest</code> . The channel is expected to forward the message to the proper remote instance. The method should return <code>false</code> if the message can not be sent in case of a network disconnection.
<code>broadcastTag(tag: Tag, ref: FarReference)</code>	Invoked when TSAT wants to broadcast information on an exported object. The channel is expected to announce the object using its tag and provide a far reference to any remote instance that is trying to discover the object.
<b>Status Polling</b>	
<code>getLocalId() : string</code>	Should return an identifier for the local instance. The identifier should be unique between all instances on the same channel.
<code>getStatus () : ConnectionStatus</code>	Returns the current connection status of the channel (connected or disconnected). Uses by TSAT to know when it should buffer messages .
<b>Event Handling Methods</b>	
<code>setMsgReceivedCB(cb: ChanMsgReceivedCB)</code>	Register an event handler for incoming messages.
<code>setTagReceivedCB(cb: ChanTagReceivedCB)</code>	Register an event handler for incoming tag broadcasts.
<code>setNameUpdatedCB(cb: ChanNameUpdatedCB)</code>	Register an event handler for the discovery of new peers on the network.
<code>setChanChangedCB(cb: ChanChanChangedCB)</code>	Register an event handler for connection status changes.

Section 3.2.4 lists methods of the abstract `Channel` class that need to be implemented. Direct message support can be implemented using the `sendMessage` method, which takes a message and a far reference as the destination. The channel object should serialise the message appropriately for the channel and relay it to the correct peer based on the information from the far reference object. To ensure unique identifications of peers within a channel, `getLocalId` should be implemented to return a unique identifier per `Channel/VM` instance. For example, in a P2P network, a unique identifier can be created using the IP address of the channel socket server.

Broadcast message support, which is mainly used for discovering objects on a channel, can either use the main direct message channel itself or another communication mechanism. It should be implemented by defining the `broadcastTag` method. As an example of a possible channel configuration, the P2P channel uses UDP multi-casting for message broadcasts and TCP sockets for direct messages.

### 3.3 CRDT and Replication in Flec

Flec provides an extensible programming interface for developing and using CRDTs. As explained in Section 2.2, CRDTs are the most well-known family of strong, eventually consistent (SEC) data types. It relies on the underlying TSAT framework for discovering and communicating with replicas in a distributed dynamic network. Flec provides constructs for generic state-based, operation-based, and pure-operation-based CRDTs. These are built on an open implementation, where internal framework details can be accessed and modified through a Meta-Object-Protocol (MOP) layer, which reifies most distribution and replication mechanisms. Since many CRDTs rely on causal ordering to simplify the design of CRDT implementations, Flec integrates a reliable causal broadcasting [Lam78] layer and reifies key interaction points as part of the open implementation.

#### 3.3.1 Core Replication Algorithm

Flec’s CRDT framework provides abstractions for replica discovery and data replication. The framework uses TSAT’s discovery and communication over far-references.

Section 3.3.1 summarises the interface provided by the Core CRDT API. These can be used to implement custom CRDTs and form the basis for state-based, operation-based, and pure-operation-based CRDT frameworks.

---

**Algorithm 5:** performOperation in the Core CRDT API.

---

```

Input: an operation  $o$ , with arguments  $args$ 
 $this.doOperation(o, args)$ ;
foreach  $replica$  in  $replicas$  do
  |  $replica \leftarrow doOperation(o, args)$ ;
end

```

---

The `performOperation` is the operation that is responsible for propagating operations to other replicas. Algorithm 5 shows a simplified pseudocode representation of the internal logic.

Table 3.3: Core CRDT Interface in Flec.

---

<b>Flec Core CRDT Interface</b>	
<hr/> Initialisation Methods <hr/>	
<code>goOnline(actor, tagName)</code>	Register the CRDT with an actor to enable its discovery by other peers on the network using the specified tag name.
<hr/> Event Handling Methods <hr/>	
<code>onOperation(op, args)</code>	Invoked whenever a new update arrives (both from a local and remote source).
<code>onLoaded()</code>	Invoked when a new CRDT object finishes initialising.
<code>onNewReplica(ref, refs)</code>	Invoked when a new CRDT replica is discovered.
<code>setUpdateCallback(cb)</code>	Register an event handler that will be invoked when the CRDT state has been updated.
<hr/> Command Interface <hr/>	
<code>performOperation(op, args)</code>	Performs an operation on the receiver CRDT object. <code>op</code> is an enumerable type, and together with the <code>args</code> variable, it represents the operation that has to be applied. It will cause the <code>onOperation</code> method to be invoked first locally and then remotely (by means of message propagation).
<code>generateUniqueId()</code>	Generate a new network unique ID.

---

It starts by invoking the `doOperation` locally and then asynchronously applying it to all discovered replicas. The arrow operator ( $\leftarrow$ ) represents a message send to the internal far reference for the remote replica.

### 3.3.1.1 Guarantees

Flec, by default, ensures that all operations from a particular Actor are transferred in FIFO order, with exactly-once arrival semantics. Flec assumes a fail-and-recover failure model by default, where messages that cannot be delivered will be buffered until the recipient node returns online. It is possible to change this configuration with timeouts and resource leasing. However, we will assume this model for the following sections and chapters. Regarding CRDTs, this means that all operations from a certain replica will always arrive in the order that they were sent, operations will never be duplications, and as long as nodes are functioning correctly, all operations will be replicated to all replicas.

### 3.3.1.2 Implementing a State-Based Counter CRDT in Flec

We will now detail the core CRDT API through the state-based Positive-Negative Counter (PN-Counter) [SPBZ11a] explained previously. Recall from the specification in Section 2.2.1.1 that a PN-Counter is a replicated counter, which can be incremented and decremented.

**Using a PN-Counter in Flec.** Before we go into detail about implementing the PN-Counter CRDT, we show how it can be used by an application built on top of Flec, in listing Listing 3.9. We first initialise a new actor, which will store our replica. We then create a new PN-Counter replica by initiating the `PNCounter` class, and make the replica available on the network by using the `goOnline` primitive. `goOnline` instructs the actor to make the replica available under a particular tag name (*MyCounter* in this case). Any other replicas in the network that share the tag name will automatically be linked to the new replica. We then add a callback handler to the replica, which ensures that any new updates will be logged to the console. Finally, we perform several updates on the replica and then propagate its state to all discovered replicas (that share the same tag name).

Listing 3.9: Code snippet showing the use of a PN-Counter CRDT in Flec.

```

1 ctx.newActor("MyActor", (actor) => {
2
3   let replica = new PNCounter();
4
5   replica.goOnline(actor, "MyCounter");
6   replica.setUpdateCallback(rep => {
7     console.log("[Counter - A] Updated to:", rep.getValue());
8   })
9
10  replica.inc();
11  replica.inc(10);
12
13  replica.propagateState();
14 });

```

**Implementing a PN-Counter in Flec.** Internally, PN-Counter replicas keep track of negative counters (for all decrements) and positive counters (for all increments) per replica. When the state of one replica is merged with another, the resulting state will have the maximum counter value for every replica counter. The value of the counter can be computed by summing all positive counter values and subtracting all negative counter values.

Listing 3.10 shows the implementation of the PN-Counter CRDT in Flec. We start by extending the `CRDT<StateOperations>` class. The `CRDT` class is a generic class that accepts a type parameter for defining the operations that replicas can receive. Recall from Section 2.2.1.1 that state-based CRDTs have a merge function for merging replica state. We specify this in the `StateOperations` enum type.

The class declares the map data structures that will be used for the positive and negative counters. The keys of these maps will be the replica id's; the values will be the associated counters.

The `inc` and `dec` implement the increment and decrement logic respectively. They modify the counter value in the counter maps based on the unique id of the replica (obtained with `this.id`). `getValue()` computes the actual counter value by summing and decrementing all the values in the corresponding maps.

## CHAPTER 3. FLEC: A PROGRAMMING FRAMEWORK FOR EVENTUALLY CONSISTENT SYSTEMS

---

Listing 3.10: Code snippet showing the implementation of a state-based PN-Counter CRDT in Flec.

```
1 enum StateOperations { Merge };
2
3 export class PNCounter extends CRDT<StateOperations> {
4   positiveCounters : Map<string, number> = new MapWithDefault(0);
5   negativeCounters : Map<string, number> = new MapWithDefault(0);
6
7   onOperation(op, args) {
8     if (op == StateOperations.Merge) this.merge(args[0], args[1]);
9   }
10
11  merge(p, n) {
12    p.forEach(([id, value]) => this.positiveCounters.set(id, Math.
13      max(value, this.positiveCounters.get(id))));
14    n.forEach(([id, value]) => this.negativeCounters.set(id, Math.
15      max(value, this.negativeCounters.get(id))));
16  }
17
18  inc(n: number = 1) {
19    const val = this.positiveCounters.get(this.id);
20    this.positiveCounters.set(this.id, val + n);
21  }
22
23  dec(n: number = 1) {
24    const val = this.negativeCounters.get(this.id);
25    this.negativeCounters.set(this.id, val + n);
26  }
27
28  getValue() {
29    let result = 0;
30    this.positiveCounters.forEach(counter => result += counter);
31    this.negativeCounters.forEach(counter => result -= counter);
32    return result;
33  }
34
35  propagateState() {
36    this.performOperation(StateOperations.Merge, [ Array.from(this
37      .positiveCounters), Array.from(this.negativeCounters) ]);
38  }
39 }
```

A propagation method is provided that will broadcast the state to all known replicas by invoking the built-in `performOperation` method with the counter values. The counter values are transformed from a map into an array containing key/value pairs, to ensure that the values are sent to remote replicas by copy, rather than by far reference.

Then, we implement the `onOperation` method and specify that if we receive a merge request, we need to apply the merge method. The arguments for the merge method will be the positive and negative counter values. The merge method computes the maximum counter values for every counter per replica and updates the state accordingly.

### 3.3.2 Operation-Based CRDT Layer

Flec provides an Operation-Based CRDT layer on top of the Core CRDT Framework. It provides features such as Reliable Causal Broadcasting (RGB), computation of causal stability for automated garbage collection, and automated state serialisation.

Section 3.3.2 lists the extended API interface of the Operation-Based CRDT class. CRDTs built on top of the API do not need to be aware of the implementation details of the causal stability algorithms and extensions to it.

**Intended Audience: Implementors vs Extenders.** In Section 3.3.2, we categorise methods based on their intended audience. We have methods categorised under *For CRDT Implementors*, and under *For Framework Extenders*. With CRDT implementors, we mean developers implementing standard operation-based CRDTs. They do not need any specific middleware behaviour and can rely on the default behaviour provided by Flec. With framework extenders, we mean developers introducing new concepts into Flec, such as delivery middleware extensions, changes to garbage collection algorithms, and more. Of course, CRDT implementors can use the methods for framework extenders if needed, but this should only be required in rare cases. We will use both types of methods throughout this dissertation to extend the framework and provide CRDT implementations.

## CHAPTER 3. FLEC: A PROGRAMMING FRAMEWORK FOR EVENTUALLY CONSISTENT SYSTEMS

Table 3.4: Operation-Based CRDT Interface in Flec.

<b>Operation-Based CRDT Interface (class: OpCRDT)</b>	
<b>Event Handling Methods (For CRDT Implementors)</b>	
<code>onOperation(clock, op, args)</code>	Invoked whenever a new update arrives. This version provides a logical clock from the RCB layer as an argument and will always be executed in causal order. This method is intended for CRDT implementors wanting to implement a particular CRDT.
<code>gcStable()</code>	Invoked after a new operation is applied and when a CRDT implementation may want to remove causally stable operations.
<b>Event Handling Methods (For Framework Extenders)</b>	
<code>onBufferedOperation(clock, op, args)</code>	Invoked when a new operation has arrived but is put in the RCB buffer as causal dependencies are missing.
<code>doOperation(clock, op, args, isLocal)</code>	Low-level operation that handles all incoming messages and ensures they are processed in causal order. This method has a default implementation which is responsible for the invocation of <code>onOperation</code> and <code>onBufferedOperation</code> .
<b>Command Interface (For CRDT Implementors)</b>	
<code>isCausallyStable(clock)</code>	Can be used to check if a certain clock is causally stable.
<code>disableCausalDelivery()</code>	Optionally disable causal delivery if not required.
<b>Command Interface (For Framework Extenders)</b>	
<code>setStable(id, iClock)</code>	Notify the underlying RCB layer that the individual clock <i>iClock</i> for a replica with id <i>id</i> is causally stable. This will update all locally stored clocks for all replicas and trigger the metadata removal mechanisms.
<code>getBufferData()</code>	Get a copy of the entries in the RCB buffer.

### 3.3.2.1 Replication Algorithm

We now further detail the replication algorithm employed by our framework based on the aforementioned interface of the op-based CRDT layer. For the sake of brevity, we only list the essential logic for extending the RCB framework with support for causal stability messages.

---

**Algorithm 6:** performOperation for operation-based CRDTs.

---

```
Input: an operation  $o$ , with arguments  $args$   
 $localClock.increment()$ ;  
 $var\ clock := localClock.copy()$ ;  
 $this.doOperation(clock, o, args)$ ;  
foreach  $replica$  in  $replicas$  do  
|  $replica \leftarrow doOperation(clock, o, args)$ ;  
end
```

---

In contrast to Algorithm 5, the `performOperation` method for operation-based CRDTs is modified to add logical timestamps to replicated operations, as can be seen in Algorithm 6. It starts by incrementing the local logical clock of the replica to which it is being applied. Then, the operation is applied locally and then propagated to all replicas. Note that the code creates a copy of the original clock, as Flec cannot ensure proper serialisation of objects when shared between *local actors*. This could mean that a receiving actor could inadvertently modify the clock. As such, we ensure that we always pass a copy of the clock. This has been fixed in the development version of Flec, which supports passing objects as explicate isolates [CMGB<sup>+</sup>07], which ensures that only copies or far references are passed to local actors.

### 3.3.2.2 Implementing an Operation-Based Replicated Counter

We now discuss the implementation of an operation-based counter CRDT, based on the specification listed in Section 2.2.1.2. Listing 3.11 details the implementations of an operation-based Counter in Flec. In lines 3-6, we first define the `Counter` interface of the CRDT, i.e. that a counter has `increment` and `decrement` operations. At line 8, we define our `OpCounter` class by extending the generic `OpCRDT` class provided by Flec. The `OpCRDT` class takes a generic type argument for specifying the replica interface,

Listing 3.11: Code snippet showing the implementation of an operation-based counter CRDT in Flec.

```
1 import { OpCRDT } from "./flec/opcrdt";
2
3 interface Counter {
4     Inc(n: number),
5     Dec(n: number)
6 }
7
8 export class OpCounter extends OpCRDT<Counter> {
9     handler: Counter;
10    value : number;
11
12    constructor() {
13        super();
14        this.value = 0;
15
16        this.handler = {
17            Inc: n => this.value += n,
18            Dec: n => this.value -= n,
19        }
20    }
21
22    inc(n: number = 1) {
23        this.performOperation("Inc", [n]);
24    }
25
26    dec(n: number = 1) {
27        this.performOperation("Dec", [n]);
28    }
29
30    getValue() {
31        return this.value;
32    }
33
34 }
```

which in our case is the `Counter` interface. By specifying the type interface, Flec will ensure the conformity of the handler functions at the type level.

In contrast to the state-based PN-Counter implementation, where we need to manually implement the merge function for the counter state, only individual operations must be defined. This is done at line 16, where we define the operation handlers. Lines 22 and 26 implement the public interface for the CRDT class; this is what is exposed to end-user code and what will be used to apply local updates. The methods simply invoke `performOperation`, which ensures the replication of the operations as defined in Algorithm 6. Flec ensures the type safety of these functions; it is not possible to call `performOperation` for operations other than those defined in the type interface (`Inc` and `Dec` in this case). Finally, the `getValue` method returns the current state of the replica.

### 3.3.3 Pure Operation-Based CRDT API

Flec provides a Pure Operation-Based CRDT framework implemented on top of the Operation-Based CRDT API. The extended API is listed in Table 3.5, which follows the specification seen in Section 2.3.1. The redundancy relations,  $R$  and  $R_{\_}$ , can be defined for CRDT implementations by implementing the `isPrecedingOperationRedundant`, `isConcurrentOperationRedundant`, and `isArrivingOperationRedundant` methods. These methods will be invoked when operations arrive, and their return value will dictate if particular operations need to be stored in the log. Operations (arriving or stored in the log) are represented through the `POLogEntry` class as defined in Section 3.3.3. This class provides an extensive interface that allows access to crucial information related to operations, such as the argument list and logical timestamps. We will now detail how these APIs can be used to implement concrete CRDTs.

#### 3.3.3.1 Implementing a Pure Operation-Based Set CRDT

This section shows how to implement an add-wins pure operation-based CRDT in Flec. The implementation is based on the specification for the pure-op AW-Set shown in Section 2.3.2. Similar to operation-based CRDTs, the set of operations that can be applied must be defined in an

Table 3.5: Pure Operation-Based CRDT Interface in Flec.

<b>Pure Operation-Based CRDT Interface (class: PureOpCRDT)</b>	
<b>Event Handling Methods (For CRDT Implementors)</b>	
<code>isPrecedingOperationRedundant</code> , <code>isConcurrentOperationRedundant</code>	Encodes the $\mathbf{R}_-$ (or $\mathbf{R}_0, \mathbf{R}_1$ ) binary relation(s); defining if existing log entries become redundant by a new operation. Alternatively, <b>isRedundantByOperation</b> unifies both methods.
<code>isArrivingOperationRedundant</code>	Encodes the $\mathbf{R}$ binary relation (i.e., is a new operation redundant by an already existing log entry).
<code>onLogEntryStable</code>	Performs an action when an operation becomes stable.
<code>onRemoveLogEntry</code>	Performs an action when a particular item is removed from the log (for example, if it was marked redundant by <code>isRedundantByOperation</code> ).
<code>onAddLogEntry</code>	Performs an action when a new operation arrives in the log.
<b>Command Interface (For CRDT Implementors)</b>	
<code>getLog</code>	Returns a list containing all current log entries.
<code>getConcurrentEntries</code>	Gets all concurrent log entries for an operation.
<b>Syntactic Sugar</b>	
<code>perform.XX(args)</code>	Syntactic sugar for <code>performOperation(op, args)</code> . For example, <code>this.perform.add("A")</code> corresponds to <code>this.performOperation("add", "A")</code> . Flec enforces at the type level that the specified operation corresponds with the type of the CRDT.

Table 3.6: PO-Log Entry API in Flec.

<b>PO-Log Entry Interface (class: PologEntry)</b>	
<b>Queries</b>	
<code>is(opType)</code>	Returns if the entry represents an operation of a certain type. For example, <code>entry.is("add")</code> can be used to check if the entry is an add operation. Flec enforces at the type level that <code>opType</code> corresponds with the possible operation types of the CRDT.
<code>hasSameArgAs(entry, idx)</code>	Can be used to compare if the arguments to an operation correspond with the arguments of another log entry.
<code>addProperty(key, value)</code>	Add custom metadata to a log entry.
<code>getProperty(key)</code>	Get custom metadata from a log entry.
<b>Causal Dependencies</b>	
<code>precedes(entry)</code>	Returns if the entry causally precedes another entry.
<code>isConcurrent(entry)</code>	Returns if the entry is concurrent to another entry.
<code>follows(entry)</code>	Returns if the entry causally follows another entry.
<code>isStable()</code>	Returns if the entry has a logical timestamp that is causally stable.
<b>Identity</b>	
<code>getOrigin()</code>	Returns the id of the replica that is the origin of the log entry.
<code>getUniqueId()</code>	Returns a string that uniquely identifies the log entry.
<b>Syntactic Sugar</b>	
<code>isXXX()</code>	Syntactic sugar for <code>is("XXX")</code> For example, <code>isAdd()</code> corresponds to <code>is("add")</code> .

interface. This is shown in Listing 3.13, with the `SetOperation` interface declaring the `add`, `remove`, and `clear` methods.

In Listing 3.12, we define the `AWSet` class for our CRDT, which extends the `PureOpCRDT` class using `SetOperation` interface. Following this, the abstract `IsPrecedingOperationRedundant` and `isArrivingOperationRedundant` from the `PureOpCRDT` class are implemented, defining the behaviour of the AW-Set. Finally, the `toSet`, `add`, `remove`, and `clear` methods provide the CRDT with its public interface. We will detail the implementation of these methods below.

**Concurrency Semantics.** The set’s redundancy relations are implemented using the `IsPrecedingOperationRedundant` and `isArrivingOperationRedundant` methods, corresponding to the  $R_{\_}$  and  $R$  relations, respectively. The relationships between log entries are defined through the `.is`, `.precedes`, `.follows`, `.isConcurrent` and `.hasSameArgsAs` methods from the `POLogEntry` class.

**User Interface.** Lines 15-22 show the implementation of the `toSet` method, which iterates over the log and serialises the state of the CRDT to a `Set` object. This is useful for end-users who may want to query the local state. Because the redundancy methods already filter out all remove and clear operations and all redundant add operations, we can simply copy all log values directly into the set object.

Finally, in lines 29-40, the mutator methods are implemented. They signal the RCB layer using `perform` (which is syntactical sugar for `performOperation`) that a particular operation has been applied, and the RCB layer will propagate and apply it on all other replicas in the system.

### 3.3.3.2 A Remove-Wins Pure Operation-Based Set

For completeness, we provide the implementation of the RW-Set, as defined in Section 2.3.2, in Appendix B.

## 3.4 Conclusion

In this chapter, we provided an in-depth exploration of the Flec programming framework, its integration with TypeScript, and the facilitation of

Listing 3.12: Pure operation-based AW-Set implementation.

```
1 type SetEntry = PLogEntry<SetOperation>;
2
3 export class AWSet extends PureOpCRDT<SetOperations> {
4   protected isPrecedingOperationRedundant(existing: SetEntry,
5     arriving: SetEntry, isRedundant: boolean) {
6     return arriving.isClear() ||
7       existing.hasSameArgAs(arriving);
8   }
9
10  protected isArrivingOperationRedundant(arriving: SetEntry) {
11    return arriving.isRemove() ||
12      arriving.isClear();
13  }
14
15  public toSet() : Set<string> {
16    let set = new Set<string>();
17
18    this.getLog().forEach(entry => {
19      set.add(entry.args[0])
20    });
21
22    return set;
23  }
24
25  public contains(element) : boolean {
26    return this.toSet().has(element);
27  }
28
29  public add(element) {
30    this.perform.add(element);
31  }
32
33  public remove(element) {
34    this.perform.remove(element);
35  }
36
37  public clear() {
38    this.perform.clear();
39  }
40 }
```

Listing 3.13: SetOperation Type Interface.

```
1 interface SetOperations {  
2     add(element: string);  
3     remove(element: string);  
4     clear();  
5 }
```

distributed programming through TSAT and CRDTs. By integrating the advantages of TypeScript and the Ambient-Oriented Programming Model, Flec emerges as a versatile solution for addressing the complexities inherent in distributed computing. We demonstrate how Flec offers an extensible programming interface for implementing CRDTs, intending to lower the barriers to utilising CRDTs in application development. This chapter's concepts and practical applications provide a foundation for exploring more advanced distributed systems using Flec. In the following chapters, Flec is applied as a laboratory for experimenting with CRDTs, allowing us to test and validate our contributions.

## Chapter 4

# Efficiently Supporting Dynamic Networks

In this chapter, we focus on the challenges that arise with Conflict-free Replicated Data Types (CRDTs) operating within dynamic network environments. Dynamic networks are systems in which nodes can join and leave at will without reconfiguring the entire networked system. We look at two key aspects: distribution and memory management for CRDTs in dynamic networks.

As mentioned in Section 2.4.1, CRDT designs (such as [SPBZ11a] and [BAS17]) typically assume a fixed network structure, which limits the applicability of CRDTs in modern distributed applications that often require dynamic scalability. For instance, collaborative platforms like Google Docs or e-commerce applications like Amazon involve unpredictable users or devices, necessitating a more adaptable approach to CRDTs in dynamic networks. Several CRDT-based systems rely on a centralised approach with a membership protocol [LPS10, LHCW18], where join and leaves are coordinated through a single server.

Earlier work on Group Communication Systems [Gol92, ADKM92] have explored support for the handling of dynamic networks in the context of eventually consistent environments, where nodes have an eventually consistent view of which other nodes are present in the group. To bring similar support for dynamic networks to CRDT platforms, without relying on central coordination, two main challenges must be addressed. First, no well-defined approaches or mechanisms exist for sharing existing states

and operations with new nodes. This is essential to ensure new nodes can properly participate in a network and eventually converge. For example, if nodes with operation-based CRDT replicas miss updates that were applied before they joined, they may end up in a faulty state that never converges. It is not enough to simply share a copy of the state from one node to new nodes, as concurrent modifications may occur during the joining process.

Secondly, traditional approaches for managing and removing metadata in CRDTs also rely on systems that assume fixed networks (see Section 2.4.2). For example, approaches that rely on causal stability for removing metadata, such as the Pure Operation-Based Framework [BAS17], do not take into account that the number of known nodes might change.

We introduce a join model that determines how new nodes can acquire the correct state and participate in an existing network without relying on a centralised server. Additionally, we provide a memory management scheme to improve metadata removal in such environments.

We conclude this chapter by evaluating our implementation and benchmarking the memory usage of replicated sets under various metadata removal techniques, including our proposed method. We demonstrate that our approach speeds up the metadata removal process and proactively identifies when an operation can no longer be concurrent, reducing memory usage more efficiently than previous methods.

### 4.1 Definitions and Assumptions

In this chapter, we assume a full-mesh peer-to-peer system (as supported by Flec) for our designs. As such, all peers have the same responsibilities and can communicate directly with each other.

We define a *node* in the system as a VM or machine that hosts a CRDT replica. We employ the term *network* as the set of nodes hosting a replica for one CRDT. We assume that every node in the network holds a single replica of a CRDT. In the case that a node disconnects, we assume this to be a transient failure [TvS06] and expect that the node will eventually recover and return to the network (following the assumptions made by Flec in Section 3.2.3.2). In other words, we assume a fail-and-recover failure model.

We also assume that the communication layer buffers messages that cannot be delivered when nodes are temporarily disconnected (again, following the asynchronous communication model of Flec).

Furthermore, we assume that eventually, all messages arrive, i.e., reliable communication with no message lost or duplication, e.g., TCP/IP, and that there are no byzantine failures, i.e., no malicious nodes. We expect that the middleware relies on acknowledgement messages to ensure message delivery and processing, which is a reasonable expectation, as TCP/IP and similar protocols also use handshakes and acknowledgements to ensure in-order delivery of messages.

## 4.2 A Dynamic Join Model for CRDTs

In this section, we define our dynamic join model for CRDTs, a model in which nodes hosting CRDT replicas can dynamically join an existing network. We will first detail our approach informally through an example setup and then provide a specification in pseudo-code.

In our join model, when a node (the *joining node*) wants to join a network, it has to do so by initiating contact with a single node (the *join node*), which is already part of the network (and hosts a replica). This can be seen in Figure 4.1 where the white  $N$  node sends a join request message to the grey node  $A$ . Grey nodes represent nodes that are fully part of the network. Dashed lines between nodes represent a 'knows' relation. In our example, nodes  $A$ ,  $B$ , and  $C$  all know each other and form the existing network, and they are all *acquaintances* of each other.

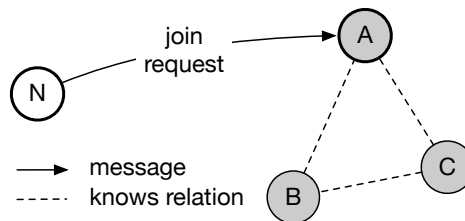


Figure 4.1: Step 1: A node requests to join a network.

When a node receives a join request, it responds by sending network information about its acquaintances to the joining node and then adding the new node to its known nodes. The shared network information should

contain all the details and information the new node needs to initiate communication with the other nodes (such as IP addresses).

The joining node will start communication with all nodes in the network and request to *link* to them (see Figure 4.2). Existing nodes will respond by acknowledging the link request, passing their current logical clocks to the joining node, and adding the joining node to their acquaintance list. Once the joining node is in the acquaintance list of some nodes, it may receive updates (of applied operations) from these nodes. The joining node has to buffer these operations until after it has received acknowledgements from all nodes in the network and a copy from the state of the join node.

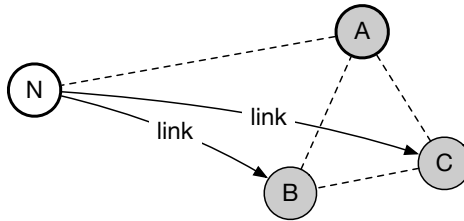


Figure 4.2: Step 2: The new node contacts all other nodes in the network.

Once the joining node has received acknowledgements from all nodes, it will request the join node for its state (visualised in Figure 4.3). It requests this together with the merged clock that it received with all the acknowledgements from all the other nodes in the network. This allows the join node to process the request in causal order, as it's essential that the join node first receives any operation that may have been concurrent with the linking process. Once this is the case, the join node will send its state to the joining node, where the joining node will store the state. All buffered operations on the joining node that are contained in this state are then discarded, and the remaining buffered operations will be applied in causal order. At this point, the joining node is a full-fledged member of the network.

### 4.2.1 Concurrent Joins

It is possible that while a node is in the process of joining a network, concurrently, another (new) node will attempt to join as well. To support this case in our approach, join nodes must relay all link requests from

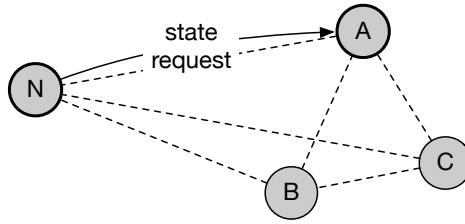


Figure 4.3: Step 3: The new node performs a state request once it has been fully acknowledged.

joining nodes to other joining nodes. To this end, consider the scenario shown in Figure 4.4, where node  $N$  is sending a join request to node  $A$  and concurrently node  $O$  is sending a join request to node  $C$ . Following the above protocol,  $O$  will receive the network state from node  $C$ . However, this state may still lack information about node  $N$ . Similarly, the network state that node  $N$  receives may lack information on node  $O$ . As both nodes may not be aware of each other they will not be able to link with each other as-is.

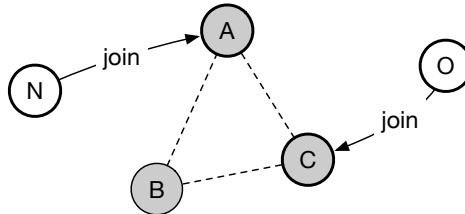


Figure 4.4: Two nodes perform simultaneously join requests to different nodes in the network.

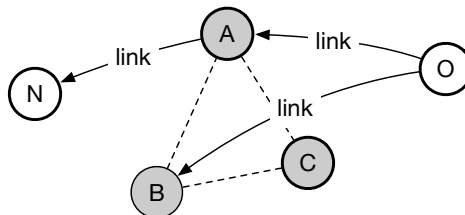


Figure 4.5: Node  $A$  forwards a link message from node  $O$  to node  $N$ .

As such, in the example in Figure 4.5, node  $A$  forwards a link request from node  $O$  to node  $N$  (the new node being handled by  $A$ ). Similarly, node  $O$  will eventually receive a forwarded link request from  $N$ .

By forwarding the link requests, nodes that join concurrently will always receive link requests from each other before they can finalise joining the network. The nodes handling the join request (e.g.  $A$  and  $C$  in Figure 4.4) will reject state requests from new nodes if they have not handled additionally forwarded links, e.g. a node is only allowed to finalise its join if it has connected to all nodes from the network state plus later forwarded link messages.

### 4.2.2 Dynamic Join Algorithm

We now provide an algorithmic specification for the Dynamic Join Model. Algorithm 7 and 8 detail the extensions that are needed to support dynamic networks, which extends on the existing CRDT replication algorithm described in Section 2.3.1 (Algorithm 4).

**State variables.** We first define the variables needed to support our join mechanism. They are listed at the start of Algorithm 7, and provide the following functionality:

- **currMode:** used to track the current joining state of the replica. It can be either *joining* or *operational*;
- **lnkNodes:** a list which tracks acquaintances (known nodes in the network);
- **pndNodes:** a list which tracks the nodes to which the current node is linking (but has not yet received an acknowledgement for).
- **joiningNodes:** tracks all nodes for which the current node is the joining node.
- **joinNode:** the node that is responsible for coordinating the join process of the current node.
- **opBuffer:** a list containing all operations that are received while the node is in the *joining* mode and not yet fully operational.

**Algorithm 7:** Core Dynamic Join Algorithm.

---

```

Data: global currMode := JOINING;
Data: global joinNode := NONE;
Data: global opBuffer := [];
Data: global lnkNodes := [];
Data: global pndNodes := [];
Data: global joiningNodes := [];
on init(node) :
  | joinNode := node; nodes := joinNode ← reqJoin;
  | lnkNodes.add(joinNode);
  | foreach n in nodes do
  |   | n ← reqLink;
  |   | pndNodes.add(n);
  | end
on reqJoin :
  | lnkNodes.add(sender);
  | joiningNodes.add(sender);
  | return lnkNodes;
on reqLink :
  | lnkNodes.add(sender);
  | sender ← ackLink(clock);
  | foreach node in joiningNodes do
  |   | node ← reqLink as sender;
  | end
on ackLink(clock) :
  | mergeClock(clock);
  | pndNodes.remove(sender);
  | lnkNodes.add(sender);
  | if pndNodes is empty then
  |   | state := joinNode ← reqState(clock);
  |   | currMode := OPERATIONAL;
  |   | foreach [t, o] in opBuffer do
  |   |   | self ← deliver(t, o);
  |   | end
  | end
on reqState(clock) with causal delivery:
  | joiningNodes.remove(sender);
  | lnkNodes.add(sender);
  | return state;

```

---

**Operation delivery during the join process.** As mentioned previously, while a node is joining, it must buffer all arriving operations. To this end, the `deliver` method from the existing CRDT framework, which is called when an operation arrives (from the local or a remote replica), is aliased to `core-deliver` (as shown in Algorithm 8). The `deliver` method is then overwritten with a new method from our join mechanism that buffers all operations in the `opBuffer` list as long as the node is not fully operational. Once the node is fully operational, operations will no longer be buffered but will be delivered through `core-deliver`.

---

**Algorithm 8:** Dynamic Join Algorithm Delivery Hook.

---

```

alias existing deliver(t, o) → core-deliver(t, o)
overwrite deliver(t, o) :
    if currMode ≠ OPERATIONAL then
        | opBuffer.push([t, o]);
    else
        | core-deliver(t, o);
    end

```

---

**Initialisation.** On initialisation (on *init*), the `init` method will add its join node to its `lnkNodes` (acquaintances) list and send a `reqJoin` request to the join node. The join node handles the `reqJoin` request by adding the joining node to the `joiningNodes` list and returning the `lnkNodes` list. The joining node will initiate contact with the nodes from this list by sending them a `reqLink` message and adding them to the `pndNodes` list.

**Linking.** Nodes that receive a `reqLink` message will store the joining node (references by the keyword `sender` in the algorithm) in its `lnkNodes` list, and acknowledge the request by responding with the `ackLink` message. These messages will also carry the logical clocks of the corresponding node. As explained earlier, link requests must be forwarded to any joining nodes. This is done by iterating over the `joiningNodes` list and forwarding the message. We specify that the message is forwarded as `sender` to ensure that the `sender` variable will still point to the original sender when received. When a joining node receives the `ackLink` message, it will first merge the clock from the responding node with its own

clock. Then, the node is removed from the `pndNodes` list and added to the `lnkNodes` list, making it a full acquaintance.

**State retrieval.** When all nodes have responded to the `reqLink` messages (i.e. the `pndNodes` list is empty), the state from the join node is requested by sending a `reqState` message. This message must be delivered through the RCB layer so that the join node processes it in causal order, to avoid any gaps in the retrieved state. The join node will remove the joining node from its `joiningNodes` list, add it to the acquaintance list, and return the state. The joining node will apply the state and any operations that have been buffered. In the algorithm, we assume that the state contains the clock of the join node and that deliver will only deliver any operations that are not present in this state (by means of comparing the clock of the state and buffered operations).

### 4.2.3 Implementing our Dynamic Join Model in Flec

To extend Flec with our join model, we benefit from the open model of the Operation-Based CRDT interface (see Section 3.3.1 and Section 3.3.2) and implement the above algorithm by extending the `onOperation`, `doOperation` and `onNewReplica` methods. The join protocol can be implemented by overriding and extending these methods, as described in the specification above. The full implementation details are described in Appendix C.

Section 4.2.3 provides a summary of the internal methods used by our implementation, and can be used by framework implementors to design new extensions.

## 4.3 Improved Metadata Removal with Eager Stability Determination

As mentioned in Section 2.3, the Pure-operation-based CRDT framework uses two mechanisms for metadata removal and memory management. The first is through redundancy relations, where logical relations are used to compare arriving operations with entries in the log. Redundant entries are removed from the log, and immediately redundant operations are never stored. The second is through the determination of causal stability. For operations that have causally stable timestamps, no new concurrent

Table 4.1: Dynamic Network Hooks for the Operation-Based CRDT API in Flec.

<b>Dynamic Network Extensions (class: <code>OpCRDT</code>)</b>	
Event Handling Methods (For Framework Extenders)	
<code>onReqJoin</code>	Handles join requests from nodes, corresponding to the <i>reqJoin</i> handler in Algorithm 8.
<code>onReqLink</code>	Handles link requests from nodes, corresponding to the <i>reqLink</i> handler in Algorithm 8.
<code>onReqState</code>	Handles state requests from nodes, corresponding to the <i>reqState</i> handler in Algorithm 8.
Command Interface (For Framework Extenders)	
<code>getNetwork</code>	Returns a list with known acquaintances (as far references). Used by <code>onJoin</code> .
<code>getState</code>	Returns the full replica state of the node, such as the PO-Log and logical clocks.
<code>setupState</code>	Applies a received state to the current replica, corresponding to the output of <code>getState</code> .

operations can arrive. As a result, the timestamps for the operation can be removed, and the operation can be compacted further.

For a node to determine that a logical timestamp is causally stable, it has to have information about all the clocks of other nodes in the network (following Definition 2.3.1). Typically, a node receives updated clock information when operations are received from other nodes. However, if certain nodes stop issuing updates for an extended period of time, this information will become stale. As a result, it may become impossible to determine causal stability.

We propose to take advantage of the RCB layer’s reliable delivery mechanism to determine causal stability eagerly without needing to wait for updates from all nodes. When an operation is applied to a replica, the underlying replication mechanism will ensure that it is (eventually) broadcasted to all other replicas. Reliable delivery requires all receiving replicas to acknowledge the reception of the operation, as shown in Figure 4.6.

In our approach, we take advantage of this design: if acknowledgements have been received from all replicas, it follows that no new operations can be concurrent to it and as a result, the operation is causally stable. As such, using this method, nodes that issue operations are able to determine causal stability for these operations if they have received acknowledgements from all nodes. To share this knowledge with other nodes, we propose to send this information to other nodes periodically, asynchronously as *stability messages* (as can be seen in Figure 4.7).

This strategy allows for determining causal stability even when some replicas are not issuing updates. This benefits situations where memory might be scarce but introduces additional network overhead as stability messages must be propagated. To enable a flexible trade-off between memory consumption and network overhead, developers can control the intervals at which stability messages are sent. Between the intervals, our approach will simply rely on the RCB middleware’s causality information to deduce causal stability, expanding on the pure-op framework’s existing meta-data removal capabilities.

#### 4.3.1 Eager Stability in Dynamic Environments

The approach described above does not yet assume a dynamic environment. We will now describe how causal stability can be correctly deter-

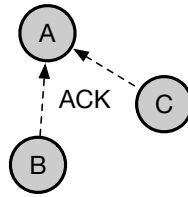


Figure 4.6: Acknowledgements used by the RCB layer to ensure reliable delivery.

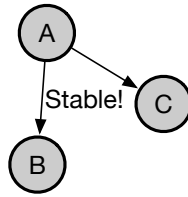


Figure 4.7: Letting other replicas know that an operation is stable.

mined in dynamic networks, combining the techniques from the previous sections.

To determine causal stability, we need causality information from *all nodes*. When a network changes in size, the meaning of what *all nodes* compose changes. If operations are issued while a node is joining, ensuring that causal stability is consistently determined over all nodes is crucial.

We propose an extension to the RCB layer to handle these cases directly. When a join node adds a new joining node to its acquaintances list, it must notify the RCB layer to expand its logical clocks with an entry for the new node. As a result, the causal stability determination mechanism on the join node will only determine causal stability if it has causal information for all nodes, which now include the new node.

Other nodes on the network will receive these expanded vector clocks as part of operations updates (or eager stability messages) from other nodes. If a node detects that a message from another node has a logical clock with an entry for an unknown node, it must expand its own clocks to contain this new entry. Consequently, it can only determine causal stability if it receives causal information from the node related to the new entry.

When a node joins a network, at any point in time, there will be at least one node (the join node) that knows the joining node. As a result, all

nodes that issue operations concurrently while another node is joining will always receive a minimum of one message from another node that contains an expanded logical clock before they can determine causal stability.

## 4.4 Implementing Eager Stability Determination in Flec

In this section, we describe the extensions to Flec needed for causal stability determination using our proposed strategy. Flec implements reliable delivery (through the RCB layer) for the operation-based CRDT framework, so we can easily piggyback on these properties.

As mentioned in Section 3.3.2, the framework provides abstractions that allow developers to propagate and receive operations while abstracting away the details of RCB and stability determination mechanisms. We now further detail the extended replication algorithm. For the sake of brevity, we only list the essential logic for extending the RCB framework with support for causal stability messages.

### 4.4.1 Extending the Replication Mechanism

---

**Algorithm 9:** Extended `performOperation` implementation.

---

```
Input: an operation o, with arguments args  
localClock.increment();  
var clock := localClock.copy();  
this.doOperation(clock, o, args);  
foreach replica in replicas do  
  | replica ← doOperation(clock, o, args);  
end  
when all operations are reliably delivered do  
  | this.notifyStable(clock);  
end
```

---

Recall from Section 3.3.2.1 the `performOperation` method from the operation-based CRDT API in Flec. This method applies an operation on a replica, typically invoked by concrete CRDT implementations, and

ensures it is to all other replicas, as shown in Algorithm 6. Algorithm 9 extends `performOperation` with support for causal stability messages.

In Algorithm 6 and 9, the method starts by incrementing the local logical clock of the replica it is being applied. The operation is then applied locally and subsequently propagated to all replicas. Algorithm 9 provides an extension with a conditional test that invokes `notifyStable` with a copy of the clock of the operation when all messages have been reliably delivered.

---

**Algorithm 10:** `notifyStable` implementation.

---

```

Input: a logical clock clock
Data: global pendingStable := False;
Data: global stableCounter := 0;
Data: global stableMsgInterval := 10;
Data: global pendingClock;
var notifyClock := localClock.copy();
notifyClock.setClockAt(clock.getId(), clock.localValue);
this.setStable(clock.getId(), clock.localValue);
pendingStable := ((this.stableCounter++) mod
    stableMsgInterval) != 0 ;
if pendingStable == True then
    | pendingClock := notifyClock;
else
    | performStableMsg(notifyClock);
end

```

---

**Broadcasting stability messages.** Algorithm 10 describes the implementation of the `notifyStable` method, which is invoked by `performOperation` after reliable delivery of an operation to all replicas. `notifyStable` will mark the local clock of the associated operation as stable through an invocation to `setStable`. Recall from Section 3.3.2 that `setStable` will trigger our framework’s internal metadata removal mechanisms by updating all the locally stored clocks for replicas.

Following this, all replicas must be notified that the clock value is stable, which we do through *stability messages*. Our approach allows these notifications to be delayed by a particular interval (defined by the

number of operations). This enables fine-grained control of the tradeoff between eager metadata removal and network resource usage.

The notifications are broadcasted through the `performStableMsg` helper function, as defined in Algorithm 11. Stability messages are delivered through the RCB layer to ensure they are ordered after all operations that were concurrent with the stable operation. This avoids cases where stability messages arrive before the other concurrent operations, which could lead to inconsistencies. While stability messages need to respect causal order, they do not need to be delivered reliably, as losing stability messages is not a critical problem. In our concrete Flec implementation, we instruct the receiving replica that no acknowledgement is needed for stability messages to avoid useless network consumption.

When a remote replica receives a stability message, `setStable` will be used to mark the associated clock as stable. This is the only required handling for stability messages, as the underlying clock changes will trigger all existing metadata removal algorithms from the framework, and no other invocations to event handlers will be needed.

---

**Algorithm 11:** `performStableMsg` implementation.

---

**Input:** a logical clock *clock*  
**foreach** *replica* in *replicas* **do**  
    ⊥ *replica* ← `doOperation(clock, STABLE, [])`;

---

**Tuning stability messages.** To enable better control over when stability messages are sent, we provide an additional method `performPendingStableMsg` that can be invoked by CRDT implementors. This method will force any pending stability messages to be sent. This allows developers to use custom heuristics to trigger the stability messages in combination with the message interval. The implementation details can be seen in Algorithm 12. Note that our framework will never call this function; it is only provided to give developers extra flexibility.

---

**Algorithm 12:** `performPendingStableMsg` implementation.

---

```
if pendingStable then
  performStableMsg(pendingClock);
  pendingStable := False;
```

---

## 4.5 Evaluation of Eager Stability Metadata Removal

In this section, we validate our extended framework with eager stability determination by running several performance experiments on Flec that aim to answer the following questions:

- **RQ1:** what is the benefit of stability messages on the log size of replicas?
- **RQ2:** what overhead does the use of stability messages incur?
- **RQ3:** what is the impact of using the log size as a heuristic for triggering the stability messages?

### 4.5.1 Experiments

We ran our experiments on a notebook machine with the following hardware specifications and software versions:

CPU	2,7 GHz Quad-Core Intel Core i7 (I7-8559U)
Memory	16 GiB
OS	macOS 13.6
Node.js	v21.0
TypeScript	v5.3

For our experimental setup, we are running several Flec actors on the machine, with the instances configured as replicas of each other, hosting a set data structure. The exact number of instances is dependent on the experiment. Flec is running on top of Node.js and is compiled using TypeScript. We then repeatedly perform operations on each replica and, depending on the experiment, either measure their log sizes or analyse the log contents. Our results are not platform-specific since we evaluate

## 4.5. EVALUATION OF EAGER STABILITY METADATA REMOVAL

---

the log size rather than memory usage. This allows us to evaluate the different extensions clearly.

In particular, in our experiments, we benchmark the Remove-Wins set (RW-Set) as described in Section 3.3.3.2. An RW-Set cannot solely rely on the redundancy mechanisms of pure operation-based CRDTs and requires compaction through causal stability to limit its log size, making it ideal for our experiments. We perform the evaluation by comparing Flec with our eager stability approach mechanism in disabled mode and then with the mechanism turned on.

### 4.5.2 Methodology

Algorithm 13 shows the core logic for benchmarking. For every benchmark, we configure the properties of the set replicas (this happens in `setupReplicas`), enabling/disabling stability messages, tweaking the size of the message interval, or setting up a log-size dependent trigger. A total of `TOTAL_ROUNDS * ROUND_SIZE` items will be added to the set replicas, where the source replica (the replica to which the set operation is directly applied) is changed every round. After every add operation, statistics regarding the first replica's log size will be measured.

Since the analysis code and CRDT implementations are deterministic, we do not need to perform multiple measurements. All the numbers from the experiments can be exactly reproduced, so there is no need for multiple runs from which confidence intervals are computed.

### 4.5.3 Assessing Meta-Data Removal for the Pure Op-Based Framework without Eager Stability

For our first test, we examine the behaviour of the vanilla pure-op RW-Set implementation, where no eager stability is used, to set a baseline for the following experiments. We test a system with 2, 4, and 8 replicas. As our methodology explains, we repeatedly keep adding items to the set replicas. To ensure that every replica eventually performs an update and that we can determine causal stability, the source node for the operations is switched every 100 operations. For example, the first 100 operations will be performed on set 0, the next 100 on set 1, and we will return to set 0 once we pass the last set. Every operation on one replica will be propagated to the other replicas in the system.

---

**Algorithm 13:** Core logic for RW-Set benchmarking.

---

```

Data: sets
var current_set := 0;
var step := 0;
setupSets();
while step < TOTAL_ROUNDS * ROUND_SIZE do
  sets[current_set] ← add("element" . step);
  takeMeasurements();
  if step mod ROUND_SIZE == 0 then
    | current_set := (current_set + 1) mod
    | NUMBER_OF_SETS;
  end
  sleep(STEP_TIME);
end

```

---

We show the results of our test in Figure 4.8. The plot shows a clear zig-zag pattern in the results: only for every 100 operations, when the source replica is switched and an update has been pushed from the new source replica, can the CRDT remove elements from the logs. This is because only at that point does new causality information about earlier operations become available, which may be enough for some replicas to determine causal stability. This also explains the initial slope in the graphs: a replica can only start determining stability once it has received updates from all other replicas. The graph shows that this happens for the first replica after the 101st operation in a system with 2 replicas. This is after the 301st and 701st operation for systems with four and eight replicas, respectively. From that point on, every 100 operations the system can determine causal stability for operations issued  $((NR\_OF\_REPLICAS - 1) * 100)$  to  $((NR\_OF\_REPLICAS - 2) * 100)$  operations earlier, which implies that the log will always be at least the size of the number of operations issued afterwards.

Note, however, there is an apparent exception to that trend in the graph: there is a slightly larger dip in the log size every  $(NR\_OF\_REPLICAS - 2)$  switches. To understand what exactly is going on, we plot a dissected view of the log for the system with 4 replicas in Figure 4.9. Each colour in the graph represents the source set for a

#### 4.5. EVALUATION OF EAGER STABILITY METADATA REMOVAL

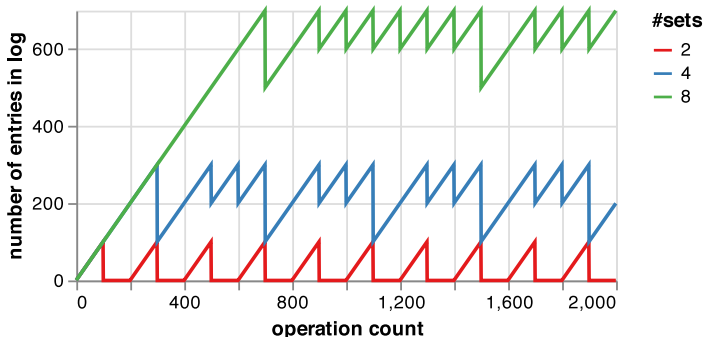


Figure 4.8: Numbers of entries in log of a pure-operation based Remove-Wins set, as operations are being applied to the sets in the system. For every 100 operations, the source replica is changed. Measurements taken for a system with 2, 4, and 8 replicas.

particular entry in the log. For example, a light blue entry means the log contains an entry for which its operation originated in set 1. For set replicas 0, 2, and 3, it takes 300 operations before the items can be removed from the log; for set 1, it only takes 200 operations.

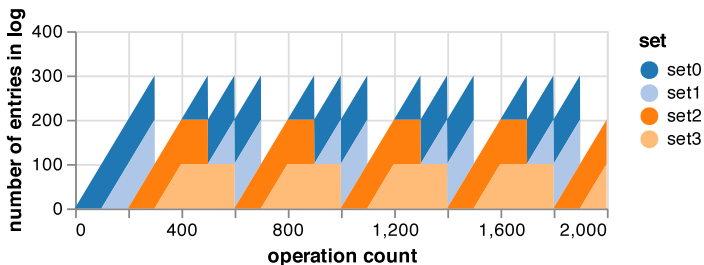


Figure 4.9: Numbers of entries in log of a pure-operation based Remove-Wins set, as operations are being applied to the sets in the system. For every 100 operations, the source node is changed. Measured in a system with 4 replicas. The colours represent the source of the entries in the logs.

The apparent exception is that since set 1 directly follows set 0 (which we are measuring), set 0 only needs causal information from sets 2 and 3 to determine causal stability for the operations issued by set 1. This information becomes available after the next two replica switches. For all the other replicas, there is always one extra node in between. For

example, for operations from set 3, we first have to go through sets 0, 1, and 2 before enough causal information is available.

**Conclusion:** These experiments show a large impact on the log size when replicas do not regularly push out updates, one that grows with the number of replicas in the system. The results confirm the missed memory optimisation opportunities from which the vanilla causal stability algorithm used in pure-op CRDTs suffers.

#### 4.5.4 Assessing the Benefits of Stability Messages

To answer RQ2, we conduct an experiment to validate (in terms of log size) the benefits of a pure operation-based framework with eager stability determination.

In particular, we compare three different setups of our framework:

- RW-Set replicas without any stability messages, meaning the framework only relies on causality information of propagated messages to deduce causal stability (no acks).
- RW-Sets replicas with our extension for stability messages; interval set to 10 operations (int=10).
- RW-Sets replicas with our extension for stability messages; interval set to 50 operations (int=50).

In all of these setups, we use four replicas, which means that each experiment is performed under circumstances identical to those from the previous section.

Figure 4.10 shows the result of this experiment. A clear drop in the log size can be observed when utilising stability messages, demonstrating their effectiveness. A smaller jigsaw pattern is visible, with drops every 10 or 50 operations, depending on the setup. There is no initial slope because stability messages are more frequent and do not depend on multiple replicas communicating.

The log still shows a slight build-up of operations when using stability messages; this is due to the nature of the benchmarking setup. To properly explain this behaviour, we plot the log sizes of all replicas in Figure 4.11. Concretely, it shows log sizes for the int=50 case. Because we switch

## 4.5. EVALUATION OF EAGER STABILITY METADATA REMOVAL

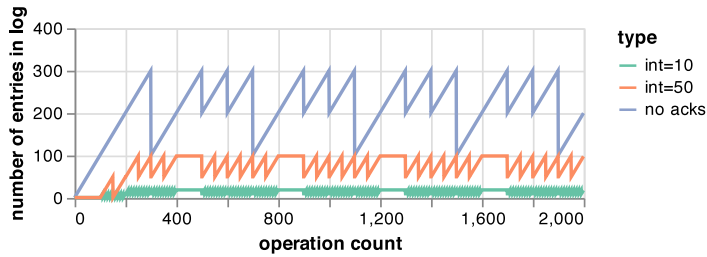


Figure 4.10: Comparison of the number of entries in the log of a pure operation-based Remove-Wins set in a system of 4 replicas, with no additional stability messages, stability messages every 10 operations, and stability messages every 50 operations. For every 100 operations, the source node is changed.

replica every 100 operations and only send stability messages after an interval of 50 messages (meaning, after 51, 101, 151... operations), the last 50 operations from the previous iteration will remain in the log.

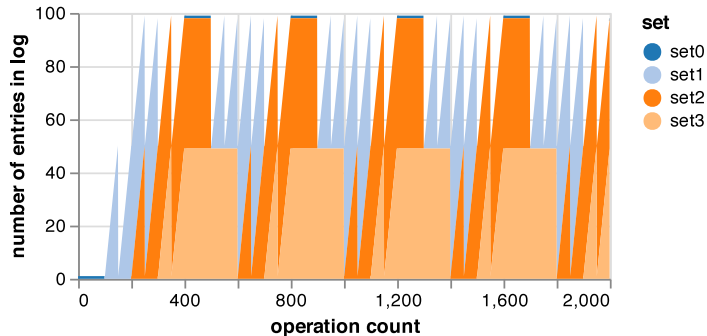


Figure 4.11: Detailed view at the number of entries in log of a pure-operation based Remove-Wins set in a system with 4 replicas and stability messages every 50 operations.

Figure 4.12 shows the results of an experiment with the same three setups but in which we change the source node every 200 operations. The initial slope of the RW-Set without stability messages has doubled, while the two instances with causal stability messages remain stable. The experiments confirm that our approach brings large memory improvements.

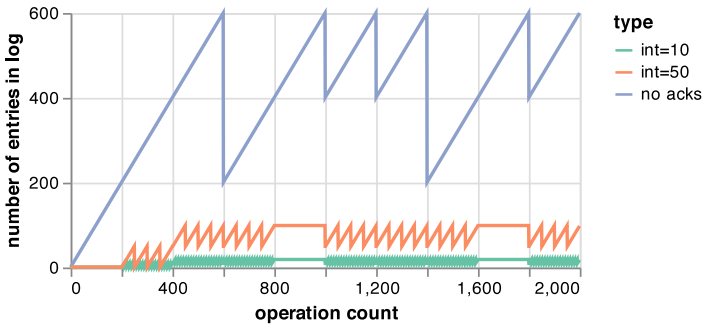


Figure 4.12: Same comparison as in Figure 4.10, but with the difference that the source node is changed every 200 operations.

**Conclusion:** These experiments answer RQ1 by showing a great benefit with the use of stability messages. The results show that our approach brings improvements when compared to the pure operation-based CRDT framework.

### 4.5.5 Assessing the Network Overhead

In this section, we assess the network overhead that our approach incurs, in order to answer RQ2. As described in Section 4.3, we use stability messages for announcing causal stability, resulting in increased network usage (as in bandwidth consumption). Figure 4.13 shows the total number of messages sent when using the same initial setup as in the previous experiment (following Figure 4.10). As shown in the graph, the overhead decreases when increasing the interval, i.e. the longer the interval is, the less overhead. This illustrates the trade-off between network (i.e. the number of stability messages sent to the network) and memory usage (i.e. the size of the log).

In our implementation we utilise separate messages per replica, meaning that the total number of messages (including those for propagating operations) in a system is relative to both the number of operations applied and to the number of replicas. Figure 4.14 shows the result of the previous experiment repeated but with 8 replicas instead of 4. Instead of showing the total number of messages sent, we now show the difference in the total number of messages sent in our approach when compared to the baseline of no stability messages. As expected, when compared to the pre-

## 4.5. EVALUATION OF EAGER STABILITY METADATA REMOVAL

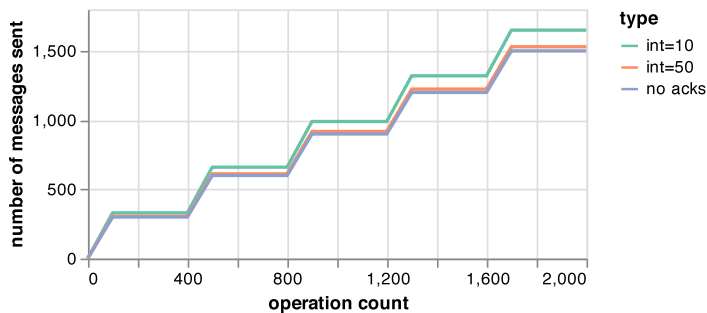


Figure 4.13: Comparison of the total number of sent messages for a pure-operation based Remove-Wins set in a system of 4 replicas, but with no additional stability messages, stability messages every 10 operations, and stability messages every 50 operations. For every 100 operations, the source node is changed.

vious experiment, the overhead has increased as more stability messages have to be sent to different replicas.

This extra overhead is unavoidable in networks where replicas can only talk directly with each other. In networks where multicasting is possible, the overhead and message sending can be dropped dramatically as all replicas can be addressed in one go.

**Conclusion:** This experiment answers RQ2, confirming that there is a network overhead when using eager stability determination but that it may be acceptable as a tradeoff with the improved memory consumption.

### 4.5.6 Assessing the Benefits of Using Log Size as a Heuristic for Stability Messages

Finally, to answer RQ3, we assess the impact of using the log size as a heuristic for triggering the stability messages, in addition to the interval-based approach. This approach may be useful to cope with the build-up of stability messages that was observed in the previous experiments (as shown in Figure 4.11). Additionally, it can be used by developers to improve the tradeoff between network and memory usage.

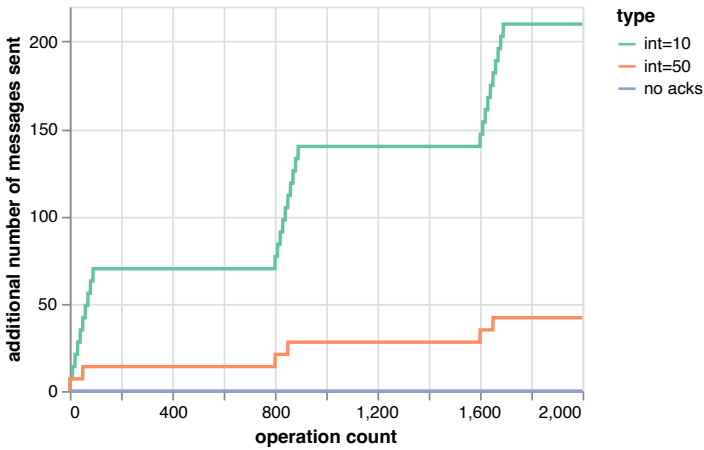


Figure 4.14: Comparison of the number of additional messages sent in a pure operation-based Remove-Wins set across a system of 8 replicas, evaluating the impact of stability messages sent every 10 operations and every 50 operations against a baseline with no stability messages. The source node is changed every 100 operations.

Figure 4.15 shows the effect of using the log size as a heuristic when enabled for the RW-Set. Again, we are using the same setup as the previous sections but with the following additions:

- In the instance where the interval is set to 10 operations, we put the trigger on 15 log entries.
- In the instance where the interval is set to 50 operations, we put the trigger on 75 log entries.

In the graph, we observe that the log can still grow to be larger than the trigger limit. The reason for this is that replicas can only broadcast stability messages for operations that they initiated. Consequently, replicas may reach their limit by receiving operations of other replicas, and they will not be able to remove these entries until they receive stability messages. Figure 4.16 shows this more clearly as it depicts the case where the trigger is set to 75 (and the interval is 50).

**Conclusion:** To conclude, our experiments show that allowing custom heuristics, such as a trigger on the log size, can improve memory efficiency

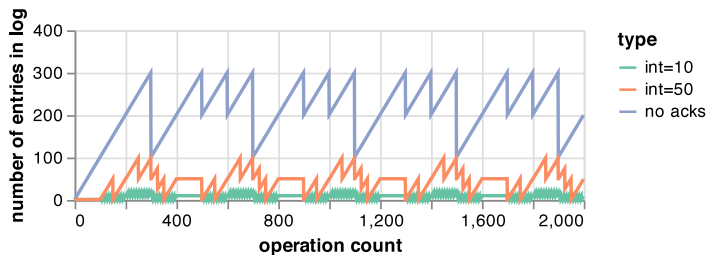


Figure 4.15: Same comparison as in Figure 4.10, but with the difference that nodes will additionally try to trigger stability messages if the number of entries log exceeds a certain size (75 entries for the system with `int=50` and 15 entries for the system with `int=10`). For every 100 operations, the source node is changed.

and allow developers to fine-tune the tradeoffs according to the data type. In general, as all replicas will receive updates and eventually hit the trigger limit, they will push out stability updates. The overall log size drops to about half of the consumption of what it was in previous experiments without this heuristic, with limited build-up.

## 4.6 Notes on Related Work

**Dynamic Networks.** As explained in Section 2.4.2, several existing libraries such as Yjs [Yjs] and Lasp [MVR15] do provide support for dynamic networks, but to the best of our knowledge, have not described the work in publications. Recently, a novel join mechanism was explored by Younes [You22], which takes a very similar approach to our work. New nodes can join a network by contacting existing members. Similar to our mechanism, they will then obtain the initial network information, contact other members, and slowly obtain a full state. The approach also allows nodes to leave through a coordination mechanism. The leaving node notifies all network nodes and then waits until this request is causally stable. The work is made available as part of a PhD dissertation and is expected to be published soon.

**Eager Stability.** Early work from [WB84] describes a replicated log that uses logical timestamps to track the causal of operations and opti-

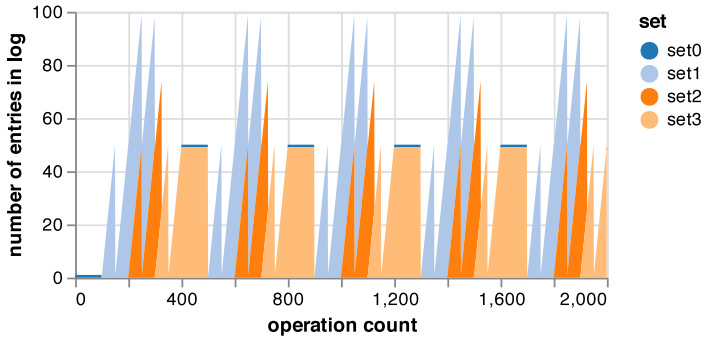


Figure 4.16: A detailed look at the number of entries in the log of a pure operation-based Remove-Wins set, in a system with 4 replicas and stability messages every 50 operations and when the number of entries in the log exceeds 75. For every 100 operations, the source node is changed.

mises the amount of data that needs to be shared. They compare their approach to related work, argue that a lack of updates may be problematic, and reason about periodic updates and the tradeoff between using logical clocks to optimise the log and the resulting communication overhead.

## 4.7 Conclusion

In this chapter, we have studied the challenges of employing Conflict-free Replicated Data Types (CRDTs) in dynamic network environments. More particularly, we propose a join model to allow a dynamic number of replicas and introduce a novel mechanism to eager determine causal stability, improving causal metadata removal.

Our join model enables support for CRDTs in dynamic networks, where peers can join at any moment. The model ensures that new nodes can acquire a correct replication state, allowing them to participate effectively in the replicated system. Our memory management techniques allow for a faster metadata cleanup process and allow systems to have a lower memory consumption. Additionally, we enable system implementors to fine-tune these processes, allowing for a balance between network resource usage and memory consumption.

We demonstrated both approaches as extensions built on Flec and evaluated the memory management techniques through several experiments. In conclusion, we show that our approach improves CRDTs' adaptability to dynamic networks and optimises their efficiency.



## Chapter 5

# Improving the Reactivity of CRDTs

Previously, we explored how to support networks with a dynamic amount of replicas. We showed how we could additionally improve memory resources by eagerly removing causal metadata. As part of our approach, we relied on RCB middleware to ensure causal ordering and reliable delivery. This is in line with most CRDT approaches and designs, such as the Pure operation-based CRDTs framework, where the causal ordering of operations is a given. Using an RCB layer typically simplifies the definition of concurrency semantics, metadata usage and removal.

However, while the benefits of using an RCB middleware are large, relying on causal ordering may not always be desirable, as it may hamper the reactivity of operation-based CRDTs [BFG<sup>+</sup>12]. When operations arrive out of causal order (e.g., before other operations that happened before), the RCB middleware buffers them until all causal predecessors arrive. Since the happened-before relation does not always imply an actual dependency between operations, operations may needlessly be buffered by the RCB middleware. This results in a less responsive CRDT, where replicas may have to wait for unrelated updates from other replicas before they can apply already received updates. This, in turn, will hamper user experience as applications may suffer from unnecessary delays. Equally important, waiting can also impact removing redundant log entries in pure operation-based CRDTs, leading to higher memory consumption.

We propose extending the pure operation-based CRDT framework with novel redundancy relations that reify information operations stored in the causal buffer (i.e., operations with missing causal dependencies). This would allow CRDT implementors to react to operations with missing dependencies without waiting for those dependencies to arrive and improve the *reactivity* of CRDTs where possible.

## 5.1 The Need for Reactive CRDTs

To demonstrate the implications of RCB on the delivery of operations, consider a sequence of operations applied to three set replicas: A, B, and C. Figure 5.1 visualises the connectivity between the replicas. Black lines denote bidirectional connectivity between replicas, and dotted lines show temporal network failures. In this case, updates are not propagated between replicas A and B.

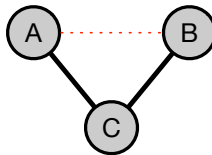


Figure 5.1: Network connectivity between set replicas.

Assume that the replicas host two different CRDTs, both with add-wins semantics: a pure operation-based add-wins set (confirming to the definition in Section 2.3.2) and an operation-based OR-Set CRDT (confirming to the definition in Section 2.2.1.3). Recall that an OR-Set CRDT does not require RCB middleware, as it uses tombstones and unique IDs to track causality.

Consider the following sequence of operations applied to both CRDTs:

1. Add(X) on replica C
2. Add(Y) on replica C
3. Add(Z) on replica B
4. Remove(X) on replica C

Table 5.1 shows the operations applied to the pure op-based Add-Wins set, while Table 5.2 shows the operations applied to the OR-Set CRDT. We apply the operations to the replicas in the following order:

Table 5.1: A sequence of operations applied to the three add-wins pure operation-based set replicas. **Replica X** ::  $\text{Op}(y)$  denotes the application of an operation  $\text{Op}$  with arguments  $y$  on **Replica X**. The last remove does not have any immediate effect on A, as A is waiting for B before it will apply any other operation from C.

Operation	Replica A	Replica B	Replica C
	{}	{}	{}
Replica C :: Add (X) Replica C :: Add (Y)	{X,Y}	{X,Y}	{X,Y}
Replica B :: Add(Z)	{X,Y}	{X,Y,Z}	{X,Y,Z}
Replica C :: Remove(X)	{X,Y}	{Y,Z}	{Y,Z}

Table 5.2: A sequence of operations applied to the three classic OR-Set replicas. **Replica X** ::  $\text{Op}(y)$  denotes the application of an operation  $\text{Op}$  with arguments  $y$  on **Replica X**. The last remove is applied immediately on set A.

Operation	Replica A	Replica B	Replica C
	{}	{}	{}
Replica C :: Add (X) Replica C :: Add (Y)	{X,Y}	{X,Y}	{X,Y}
Replica B :: Add(Z)	{X,Y}	{X,Y,Z}	{X,Y,Z}
Replica C :: Remove(X)	{Y}	{Y,Z}	{Y,Z}

1. First, the elements X and Y are added to replica C. This update is propagated to all other replicas, and their state is updated.
2. Following this, the element Z is added to replica B. This update is only sent to replica C, as there is a disconnection between replicas A and B.
3. Finally, replica C removes item X, which will be observed by replicas A and B, as replica C is connected to other replicas.

In the case of the OR-Set, the item will be immediately removed on all replicas. In the case of the add-wins pure operation-based set, the operation will not be applied on replica A, as the RCB middleware will buffer the operation. The RCB middleware detects from the causality information it received along with the operation (from replica C) that A has not yet received one or more operations from B. In practice, this means that only after the connectivity issue between A and B is resolved, and replica A receives and applies the Add(Z) operation from B, the remove operation from replica C will be applied.

The OR-Set is more reactive than the Add-Wins set as it does not rely on RCB, but its implementation is still ad-hoc and more complex: unique identifiers need to be generated for every operation, and tombstones have to be kept to ensure that remove operations commute. We aim to bring higher reactivity to pure operation-based CRDTs without relying on ad-hoc techniques. This will improve the user experience of applications that rely on such CRDTs without affecting correctness and allow for improved compaction of the PO-Log.

## 5.2 Improving the Reactivity of Pure Operations-Based CRDTs

To improve the reactivity of operation-based CRDTs utilising RCB in a generic, systematic way, we propose that the buffer of the RCB middleware where these pending messages are held is made accessible (i.e. reified) to CRDT implementors as part of the framework interface. In the context of a pure operation-based CRDT, this buffer can then be used to construct a reified-pending-messages log (*RPM-Log*). The RPM-Log, like the main

PO-Log, is partially ordered and will contain gaps of missing causal dependencies. The RPM-Log complements the existing partially-ordered log (PO-Log) and the compacted sequential state to represent the full CRDT state. Queries on the CRDT state can be computed through entries from both logs and the compacted state. Furthermore, entries in the main PO-Log and the compacted state can be made redundant by entries from the RPM-Log. When the missing dependencies for entries in the RPM-Log arrive, the entries will be moved to the main PO-Log. Note that, however, entries in the RPM-Log cannot be made redundant as long as they have not yet been moved to the main PO-Log, as concurrent operations that might be affected by the operation may yet arrive.

This approach has some additional benefits in terms of memory management, in addition to increasing the reactivity of CRDTs. Since entries from the RPM-Log can cause entries from the main log to become redundant, it can be used for decreasing memory consumption whenever intermediate disconnections are common. It may be hard to determine causal stability when disconnections are common, as not all replicas will be responsive. By observing the RPM-Log, potentially redundant information can already be removed from the main log even if causal dependencies are missing.

In the next section, we detail our approach as an extension of the pure operation-based CRDT framework. We illustrate the applicability of the approach through Add-Wins and Remove-Wins sets. Following this, we describe how we implemented our strategy and the extended sets in Flec.

### 5.2.1 Extending Semantic Log Compaction with RPM-Log Support

As explained in Section 2.3.1, pure operation-based CRDTs utilise a mechanism named *causal redundancy* to prune operations from the log whenever they become causally redundant. We extend causal redundancy with a new binary relationship  $R_\beta$  that defines the relation between log entries that live in the RCB buffer (e.g. the RPM-Log segment) and the original main log. More concretely, the relation defines which operations from the non-buffered log become redundant when new entries arrive in the RCB buffer.

The interactions between the RCB layer and the pure operation-based framework are listed in Algorithm 14, which builds on the original al-

---

**Algorithm 14:** Distributed algorithm (for a replica  $i$ ) showing the interaction between the RCB middleware and the pure operation-based CRDT framework.

---

**state:**  $s_i := \emptyset$   
**on**  $operation_i(o)$  :  
 | broadcast $_i(o)$   
**on**  $deliver_i(t, o)$  :  
 |  $s_i := (s_i \setminus \{(t', o') \mid \forall (t', o') \in$   
 |  $s_i \cdot (t', o') \mathbf{R}_-(t, o)\}) \cup \{(t, o) \mid (t, o) \mathbf{R} s_i\}$   
**on**  $buffer_i(t, o)$  :  
 |  $s_i := s_i \setminus \{(t', o') \mid \forall (t', o') \in s_i \cdot (t', o') \mathbf{R}_\beta(t, o)\}$   
**on**  $stable_i(t)$  :  
 | stabilize $_i(t, s_i)[(\perp, o)/(t, o)]$

---

gorithm listed in Section 2.3. Recall that when an operation is applied locally, it will be broadcast to all replicas. In the original definition, received operations are handled by `deliver`, always in causal order. In our approach, we reify the behaviour of the RCB layer by exposing a `buffer` function, which gets invoked when received operations are put in the RCB buffer due to missing causal dependencies. We then use both `deliver` and `buffer` for querying the  $\mathbf{R}$ ,  $\mathbf{R}_-$  and  $\mathbf{R}_\beta$  relations to check what log entries become redundant (and modify the state accordingly) and if the new operation is redundant itself. If the new operation has no missing causal dependencies and is not redundant, it will be added to the log (along with its logical timestamp). Note that the redundancy relations only affect entries in the PO-Log and that the RPM-Log is never modified. Entries are only removed from the RPM-Log when all causal predecessors have arrived.

This algorithm only describes the RCB middleware’s interaction with the pure operation-based framework. We will now describe how actual CRDTs are built on top of our now reactive pure operation-based CRDT framework.

### 5.2.2 Reactive Pure Operation-Based Sets

Table 5.3 shows an implementation for the pure operation-based add-wins set (AW-Set) CRDT using our approach. It is based on the original pure

## 5.2. IMPROVING THE REACTIVITY OF PURE OPERATIONS-BASED CRDTS

Table 5.3: Modified semantics for the Add-Wins pure-op set, supporting RPM-Log (based on approach in [BAS17]).

Framework	$(t, o) \mathbf{R} s = \text{op}(o) = (\text{clear} \vee \text{remove})$ $(t', o') \mathbf{R}_- (t, o) = t' < t \wedge (\text{op}(o) = \text{clear} \vee \text{arg}(o) = \text{arg}(o'))$ $(t, o) \mathbf{R}_\beta (t_\beta, o_\beta) = t < t_\beta \wedge (\text{op}(o_\beta) = \text{clear} \vee \text{arg}(o_\beta) = \text{arg}(o))$ $\text{stabilize}(t, s) = s$
User	$\text{toSet}(s, s_\beta) = \{v \mid (\_, [\text{op}=\text{add}, \text{arg}=v]) \in s\} \cup \{v \mid (\_, [\text{op}=\text{add}, \text{arg}=v]) \in s_\beta\}$ $\text{add}(e) = \text{operation}([\text{op}=\text{add}, \text{arg}=e])$ $\text{remove}(e) = \text{operation}([\text{op}=\text{remove}, \text{arg}=e])$

operation-based add-wins set, as described in Section 2.3.2. The table is grouped as follows: (1) functions that are used by the pure operation-based framework that dictates the interaction between new operations and entries in the log, and (2) functions that can be invoked by the user for state serialisation or mutations.

In the case of the AW-Set,  $R_\beta$  is equivalent to  $R_-$ , i.e. a (causally) older operation is redundant if it shares the same arguments with a newer operation, or if the newer operation is a clear operation. As  $R_\beta$  also encodes semantics for newer operations (albeit for buffered operations), it will typically be equivalent to  $R_-$  for most data types.

The `toSet`<sup>1</sup> function is extended to take the RPM-Log segment into account ( $s_\beta$ , the RPM-Log, is passed as an extra dependency). The fully evaluated state is the union of both the main and the RPM-Log. Finally, `add` and `remove` are applied and broadcasted to all replicas (following the definition in Algorithm 14 in Section 2.3.2).

Table 5.4 shows a modified reference implementation for the pure-op Remove-Wins set (RW-Set) CRDT using our approach, extending on the original design as shown in Section 2.3.2. Again,  $R_\beta$  is equivalent to  $R_-$ . However, unlike the AW-Set version, `toSet` cannot simply take all the adds from  $s_\beta$  operation as it needs to account for possible removes that

<sup>1</sup>In the original pure operation-based paper by [BAS17] this would be the `eval(elems, ...)` function.

Table 5.4: Modified semantics for the RW-Wins pure-op set, supporting RPM-Log (based on approach in [BGB20b, BAS17]).

Framework	$(t, o) \mathbf{R} s = \text{op}(o) = \text{clear} \vee (\text{op}(o) = \text{add} \wedge \exists(t', [\text{op} = \text{remove}, \text{arg} = \text{arg}(o)]) \in s \cdot t \sim t')$ $(t', o') \mathbf{R}_- (t, o) = (t' < t \wedge ((\text{op}(o) = \text{clear} \wedge \text{op}(t') = \text{add}) \vee \text{arg}(o) = \text{arg}(o')) \vee (t \sim t' \wedge \text{op}(o) = \text{remove} \wedge \text{op}(o') = \text{add} \wedge \text{arg}(o) = \text{arg}(o'))$ $(t, o) \mathbf{R}_\beta (t_\beta, o_\beta) = \mathbf{R}_-$ $\text{stabilize}_i(t, s)^* = \{(t', o) \mid \forall(t', o) \in s \cdot t \neq t'\} \cup \{\forall(\perp, [\text{op} = \text{add}, \text{arg} = e]) \mid (t', [\text{op} = \text{add}, \text{arg} = e]) \in s \cdot t = t'\}$
User	$\text{toSet}(s, s_\beta) = \{v \mid (\_, [\text{op} = \text{add}, \text{arg} = v]) \in s\} \cup \{v \mid (t, [\text{op} = \text{add}, \text{arg} = v]) \in s_\beta \wedge \forall(t', [\text{op} = \text{remove}, \text{arg} = v]) \in s_\beta \cdot t' < t\}$ $\text{add}(e) = \text{operation}([\text{op} = \text{add}, \text{arg} = e])$ $\text{remove}(e) = \text{operation}([\text{op} = \text{remove}, \text{arg} = e])$

$(o' \sim o)$  denote concurrent operations

\*We assume that stabilize is only called for a timestamp when all concurrent operations are stable as well.

may invalidate the add. All adds containing concurrent or newer remove for a particular element will be filtered out.

### 5.3 Implementation in Flec

In this section, we describe the implementation of our approach in Flec. We show the code extensions required for reifying buffered operations and the methods we expose for implementing reactive CRDT designs.

As shown in Section 3.3.2 from Section 3.3.2, the operation-based CRDT interface in Flec exposes a hook `onBufferedOperation` that gets called when operations are buffered by the RCB layer. We make use of this hook to implement two new constructs for the pure operation-based CRDT interface in Flec, detailed in Section 5.3, extending on Table 3.5 from Section 3.3.3.

Table 5.5: Reactive Pure Operation-Based CRDT API in Flec.

<b>Reactive Extensions (class: <code>PureOpCRDT</code>)</b>	
Event Handling Methods (For CRDT Implementors)	
<code>isRedundantByBufferedOperation</code>	Encodes the $R_\beta$ binary relation (i.e. do existing log entries become redundant by a new buffered operation).
<code>newBufferedOperation</code>	Perform an action when a new operation arrives in the RPM-Log.

Listing 5.1 shows our default implementation of `onBufferedOperation` which we use to add reactive functionality to Flec. When an operation is buffered by the RCB layer, our code will reify the operation as a `POLogEntry` object, an object that represents a log entry. As a first step, the hook will then invoke `newBufferedOperation` with the entry object as argument. This allows CRDT implementations to react to the arrival of said buffered operation in a way that is consistent with the handling of non-buffered operations (which are also represented as `POLogEntry` objects). We then, as shown from lines 5-10, iterate over the PO-Log, and invoke `isRedundantByBufferedOperation` for every log entry. Depending on the return value, log items may be removed if deemed redundant, following the behaviour of the  $R_\beta$  relation. The `removeEntry` will be called for any entry that is removed, following the existing behaviour of

Listing 5.1: A code extension to the pure op-based framework in Flec that enables reification of buffer data.

```
1 onBufferedOperation(clock: VectorClock, op: O, args: any[]){
2   const entry = new PLogEntry<O>(clock, op, args);
3   this.newBufferedOperation(entry);
4
5   for (let i=this.log.length-1; i>=0; i--) {
6     let e = this.log[i];
7     if (this.isRedundantByBufferedOperation(e, entry, false))
8     {
9       this.removeEntry( this.log[i], entry );
10      delete this.log[i];
11    }
12  }
13  this.log = this.log.filter(e => typeof e !== "undefined");
14 }
```

the framework when causally arriving operations make log entries redundant.

### 5.3.1 Implementing Reactive Sets in Flec

This section describes the implementation of the reactive add-wins and remove-wins sets (as defined in Table 5.3 and 5.4) using the reactive pure operation-based CRDT framework in Flec. Listing 5.2 shows the core implementation of the reactive add-wins set.

The implementations of `isRedundantByLog` and `isRedundantByOperation` are a 1-on-1 mapping with the described semantics for the  $R$  and  $R_{\_}$  relations shown in Table 5.3. `isRedundantByBufferedOperation`, which implements the  $R_{\beta}$  relation, is set to point to the method of `isRedundantByOperation` as its semantics are equivalent. Finally, in `toSet`, the main log (denoted by  $s$  in the table) and the RPM-Log (buffered) log (denoted by  $s_{\beta}$  in the table) are combined to determine the full state of the set.

Listing 5.3 shows the implementation for the reactive RW-Set, following the semantics in Table 5.4. Compared to the reactive AW-Set, extra code is needed to prune causally stable operations. When an entry from

Listing 5.2: Reactive AW-Set implementation in Flec.

```

1 export class ReactiveAWSet extends PureOpCRDT<SetOperations> {
2
3   // Defines R_
4   isRedundantByOperation(existing: SetEntry, arriving: SetEntry,
5     isRedundant: boolean) {
6     return existing.precedes(arriving) && ( arriving.isClear()
7       || existing.hasSameArgAs(arriving) );
8   }
9
10  //Defines Rbeta
11  isRedundantByBufferedOperation = this.isRedundantByOperation;
12
13  // Defines R
14  isRedundantByLog(entry: SetEntry) {
15    return entry.isRemove() ||
16      entry.isClear();
17  }
18
19  public toSet() {
20    const set = new Set();
21
22    this.getLog().forEach(e => set.add(e.args[0]));
23
24    this.getBufferedLog().forEach(e => {
25      if (e.isAdd())
26        set.add(entry.args[0]);
27    });
28
29    return set;
30  }
31
32  add(e) { this.perform.add(e); }
33  remove(e){ this.perform.remove(e); }
34  clear(e) { this.perform.clear(e); }
35 }

```

the log becomes stable, `setEntryStable` (line 27-32) will remove it from the log and place it in a small compacted set. `newOperation` and `newBufferedOperation` (line 34-35) make sure that this local compacted set stays up to date when the log changes. The `toSet` method (line 37-51) is also a bit more complex for the RW-Set, as it has to take the compacted set, the PO-Log and the RPM-Log into account while respecting the semantics of the data structure.

Listing 5.3: Reactive RW-Set implementation in Flec.

```
1 export class ReactiveRWSet extends PureOpCRDT<SetOperation> {
2     compactState: Set<string> = new Set();
3
4     // Encodes the R relation
5     isArrivingOperationRedundant(entry : SetEntry) {
6         return entry.isAdd() &&
7             !!this.log.find(e => e.isRemove() &&
8                 e.hasSameArgAs(entry) &&
9                 e.isConcurrent(entry));
10    }
11
12    // Partially encodes R_ for happened-before entries
13    isPrecedingOperationRedundant(
14        existing: SetEntry, arriving: SetEntry) {
15        return existing.hasSameArgAs(arriving);
16    }
17
18    // Partially encodes R_ for concurrent entries
19    isConcurrentOperationRedundant(
20        existing: SetEntry, arriving: SetEntry) {
21        return arriving.isRemove() &&
22            existing.isAdd() &&
23            existing.hasSameArgAs(arriving);
24    }
25
26    // Encodes Rbeta
27    isRedundantByBufferedOperation = this.isRedundantByOperation;
28    ... continued on the next page ...
```

```
29 // Compact entries when stable
30 setEntryStable(entry : SetEntry) : boolean {
31     if (entry.isAdd())
32         this.compactState.add(entry.args[0]);
33
34     return true;
35 }
36
37 // Ensure compacted state is kept up-to-date
38 newOperation = (entry: SetEntry) =>
39     this.compactState.delete(entry.args[0]);
40
41 newBufferedOperation = this.entry;
42
43 toSet() {
44     const set = new Set(this.compactState);
45
46     this.getLog().forEach(entry => {
47         if (entry.isAdd())
48             set.add(entry.args[0])
49     });
50
51     const sb = this.getBufferedLog();
52     sb.forEach(entry => {
53         if (entry.isAdd() && !sb.find(e => e.isRemove()) && (e.
54 isConcurrent(entry) || e.follows(entry))))
55             set.add(entry.args[0]);
56
57     return set;
58 }
59
60 add(e) { this.perform.add(e); }
61 remove(e){ this.perform.remove(e); }
62 }
```

## 5.4 Validation

In this section, we validate our extended framework with reactive CRDTs through a performance experiment on Flec. Our objective is to determine whether our approach effectively improves the reactivity of CRDTs and to quantify the difference between reactive and non-reactive CRDTs. Concretely, we aim to answer the following questions:

- **RQ1:** does our approach introduce additional overhead when delays are negligible?
- **RQ2:** does our approach remove unneeded delays and improve CRDT reactivity?

### 5.4.1 Experiments

We conducted our experiments on a notebook machine with the same configuration listed in the previous chapter’s evaluation (Section 4.5). The setup involves running multiple Flec actors on the machine, with instances configured as replicas hosting a set data structure. Flec operates on Node.js and is compiled using TypeScript.

We configure a system with three AW-Set replicas (A, B, and C). We perform two experiments, each evaluated on the system with our reactivity extensions enabled and with the extensions disabled (i.e., the standard approach).

### 5.4.2 Assessing the Overhead of Our Approach

To answer RQ1, we perform an experiment in which we perform a sequence of operations on the replicated sets in a system with no latencies. We configure all replicated sets to have the same initial state of 100 unique items. As we are using the pure operation-based CRDT framework, this implies that the PO-Log of all replicas contain 100 entries.

Every 250ms, we perform a remove operation on replica C until no items remain. Simultaneously, we add **unique items** to replica A every 250ms, with a maximum of 20 added items. All operations are instantly propagated between all replicas.

The experiment is conducted twice: once with non-reactive sets and once with reactive sets. Figure 5.2 illustrates the number of items in the

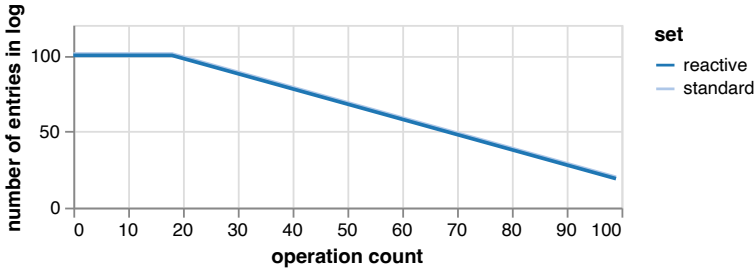


Figure 5.2: Log size of replica B over time. Measurements taken for a system with reactive AW-Set CRDTs and standard (non-reactive) AW-Set CRDTs. There is no delay between replicas A and B.

log of replica B over time for both configurations. We can observe an initial plateau where the number of entries stays stable, corresponding to the concurrent add and remove operations. After 5 seconds, the add operations stop, and the number of log entries starts dropping (as the only remaining operations are removes).

We observe that there is no difference in the number of log entries between both approaches. This is expected, as there are no latencies in the system which delay the delivery of causally dependent operations. This answers RQ1; our approach does not introduce additional overhead when a system is not experiencing delivery delays.

### 5.4.3 Assessing Improved Reactivity with Our Approach

To answer RQ2, we perform a second experiment where we make a slight modification to the system setup: we introduce a 5-second latency for updates between replicas A and B. This follows the setting we showed in Section 5.1 with Figure 5.1.

We perform the exact same experiment as previously. As there is no fixed latency between replicas A and C, operation propagation is immediate between them. Consequently, all remove operations issued by replica A will have the add operations from replica C as causal dependency. Again, the experiment is conducted twice: once with non-reactive sets and once with reactive sets. Figure 5.3 illustrates the number of items in the log of replica B over time for both configurations.

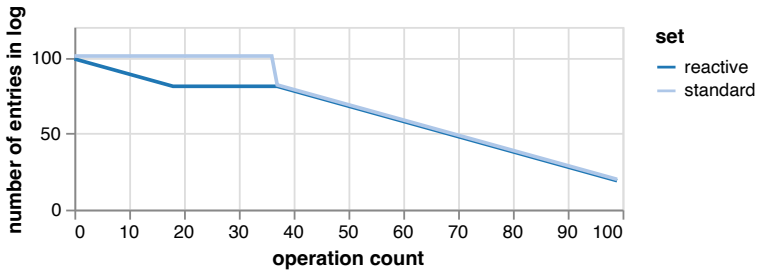


Figure 5.3: Log size of replica B over time. Measurements taken for a system with reactive AW-Set CRDTs and standard (non-reactive) AW-Set CRDTs. Operations between replicas A and B are delayed by 5 seconds.

For non-reactive sets, the log size of replica B remains fixed for the first 5 seconds as the remove operations from replica C are buffered in the RCB layer, waiting for missing dependencies from replica A. Once the add operations from replica A arrive, the removes are released and delivered to the CRDT middleware. The log size stays fixed for an additional 5 seconds (250ms x 20 add operations), as every remove is counteracted with an add. After 5 seconds (10 seconds since the start of the experiment), when the add operations cease, the log size decreases as all removals are processed. At this point, all arriving removes have no causal dependencies from other replicas and can be immediately applied.

For the reactive sets, removals are applied immediately since their effects can be computed without waiting for causal dependencies, as all the operations are on different unique elements, e.g., there no is no data dependency. This is visible in the log size, where an immediate drop is visible. With this experiment, we demonstrate that our approach allows CRDTs to achieve higher reactivity when causal dependencies for arrived operations are delayed.

## 5.5 Conclusion

In this chapter, we demonstrate how causal ordering, used in the majority of CRDT frameworks, can lead to less reactive CRDTs. We propose a mechanism to improve the reactivity of CRDTs by reifying the operation buffer of the RCB layer. Our approach is applicable to CRDT frameworks that rely on reliable causal broadcasting.

Specifically, we apply it to a pure operation-based CRDT framework, through a novel redundancy relation  $R_\beta$  that allows the comparison of buffered operations with operations stored in the PO-Log. We implement our approach in Flec, by extending on the open pure operation-based CRDT implementation. We then describe and implement extended versions of the Add-Wins and Remove-Wins sets within the extended pure operation-based CRDT framework.

We performed an evaluation that shows the effectiveness of our approach. By comparing a normal AW-Set and a reactive AW-Set in our Flec implementation, we show that reactive CRDTs can achieve higher throughput than non-reactive versions in a system experiencing delays.



## Chapter 6

# Nestable Pure Operation-Based CRDTs

This chapter explores support for nesting and composing CRDTs in a structured and systematic way. Composing CRDTs is non-trivial, as the convergence properties for CRDT designs are made to hold for *single* CRDTs and do not necessarily hold when several CRDTs are composed into a new one.

As mentioned in Section 2.4.3, recent work exploring the composition of CRDTs mainly follows a state-based design. This may result in non-sensible designs for nested CRDTs and hampers the development of CRDTs where the operation history needs to be used to improve the merging algorithm. Operation-based techniques, on the other hand, are better suited for replicating nested data structures as information on applied operations can be used to determine the optimal ordering for concurrent operations. This means it is less complex to relate different operations or even separate them when deciding what nested semantics are needed for non-commutative concurrent operations.

In this chapter, we explore a structured approach for designing and implementing nested CRDTs. CRDT designers can easily coordinate the interaction between nested structures as part of the replicated structure's concurrency semantics.

For this, we propose *Nested Pure Operation-Based CRDTs*, extending the pure operation-based CRDT framework, as described in Section 2.3, with support for nested CRDT structures. We implement nested pure

operation-based CRDTs as an extension to Flec. We validate our approach by implementing a portfolio of nested data structures and verifying it in the VeriFx language. Finally, we implement a distributed file system based on Vanakieva et al. [YYRB21] to assess the performance of our approach in comparison to a state-of-the-art JSON CRDT implementation, Automerge [KB17]. We show that the framework is general enough to nest well-known CRDT designs like maps and lists, and its network traffic performance is comparable to the state of the art.

## 6.1 Nesting Pure Operation-Based CRDTs

Currently, it is not possible to reason about nested structures within the pure operation-based CRDT framework. Redundancy relations only work on a flat level, and any logic to traverse hierarchical/nested structures has to be manually bolted on top of the framework in an ad-hoc way. More concretely, this requires developers to store nested operations in a flattened form in the main log. To evaluate and apply the log's contents, developers would need to either fully combine the logic of the nested and main top-level CRDT or encode the nested CRDT semantics in the query functions. In the former case, the redundancy relations and query functions would have to manage all concurrency rules for all needed nested strategies. This greatly complicates the design of such structures and makes them more prone to errors. In the latter case, only the query functions would need to be touched, but they would have to implement all redundancy logic from scratch. A programmer could delegate operations to separate components for the nested CRDTs, but this would ultimately imply reimplementing the delivery of operations in the query function logic, which should be kept in the framework.

In this section, we rethink nested pure operation-based CRDTs to enable the systematic construction of nested data structures. We aim to allow developers to combine and nest existing pure operation-based CRDTs and provide constructs for developing novel nested CRDTs. In particular, we focus on designs where nested structures can dynamically change at runtime, i.e., data structures that grow and shrink during an application's lifetime, such as maps and lists, where values can be CRDTs. Our approach's core idea is to offer developers constructs to define the relationship between parent and child CRDT. The framework then handles

all replication aspects regarding operations delivery in the data-structure hierarchy, ensuring that causal ordering is respected and that nested children are recursively reset when needed.

### 6.1.1 Extending the Pure Operation-Based Framework

This section describes our approach as an extension to the pure operation-based CRDT framework. We model a nested data structure as a nested hierarchy where children can be identified by a particular key and deeply nested children by an absolute path (list of keys) relative to the topmost data structure (the root CRDT). To support nested data structures, we introduce three extensions to the pure operation-based framework:

- An internal data structure to keep track of nested CRDTs (i.e., the *children* of a CRDT).
- An update propagation mechanism for nested CRDTs that delivers the applied operations ensuring that the concurrency semantics of parent data structures are upheld.
- A reset mechanism for nested CRDT operations that ensures that the concurrency semantics of children's data structures are upheld.

Each of these extensions is essential to ensure the correctness of replicated data types. In the following subsections, we elaborate on them and motivate why they are needed through several examples. In Section 6.1.2, we provide a more formal specification of our approach and extensions to the pure operation-based framework and describe example implementations for update-wins and delete-wins hash maps.

#### 6.1.1.1 Keeping Track of Nested Data Structures

Objects or data structures that have nested children typically refer to children by some key. Our approach assumes that children have a unique identifier by which they can be accessed (i.e., queried and updated). As nested children can also contain other nested elements, an absolute path can be constructed to identify a particular nested data structure, starting from the root (top-most) data structure.

At the implementation level, a CRDT developer can decide in what manner key lookup works by providing an implementation of a particular

handler function (`getChild`) that is used for lookup. The framework then provides a mechanism that allows absolute paths on a replicated structure to identify nested data structures that need to be queried or updated.

### 6.1.1.2 Updating Individual Nested CRDTs

When an operation is applied to a nested child, it is crucial that the concurrency semantics of parent data structures are upheld. Operations cannot be applied directly to children, as concurrent operations could be applied to parent nodes, some of which may even affect the keys pointing to the nested children. For example, with a hash map, an entry could be concurrently modified while it is being removed. In our approach, when an update is applied to a particular child element, we will first issue special update operations to every parent node. These update operations signal the parent CRDTs that a nested operation will be applied and that the operation should first be compared to existing log entries using redundancy relations.

To illustrate our update mechanism, consider an update-wins replicated hash map. In this case, it is important to ensure that update operations win over remove operations (on the same key). At times, the update operation itself may be immediately redundant, and as such, there is no need to propagate the operation further to a nested child.

As an example, consider Figure 6.1 showing a hash map with update-wins semantics containing nested Multi-Value<sup>1</sup> registers in three different stages. The first box (denoted by 1) shows the internal state and the PO-Log for the hash map and the register associated with the key 'B'. As explained, every update applied to the nested register has an associated update in the parent log. In this case, two concurrent updates were applied to the nested register, resulting in the state `{Hello, Hi!}`.

The second box shows the state when an `update(B, set(Hey))` is applied to the hash map. This update has a timestamp  $\langle 0, 2, 1 \rangle$  which is concurrent with some operations  $\langle 2, 0, 1 \rangle$ ,  $\langle 1, 0, 1 \rangle$ , but causally

---

<sup>1</sup>A Multi-Value register (MV-Register) [SPBZ11a] is a replicated register that, when faced with concurrent updates, will store all concurrent values. Updates that (causally) follow will replace previous values. This is in contrast to other replicated registers, for example, the Last-Writer-Wins (LWW) CRDT register [SPBZ11a] that always keeps a single value. When faced with concurrent updates, an LWW-Register will use an arbitrary method for picking a single update (such as picking the update from the replica with the highest network id).

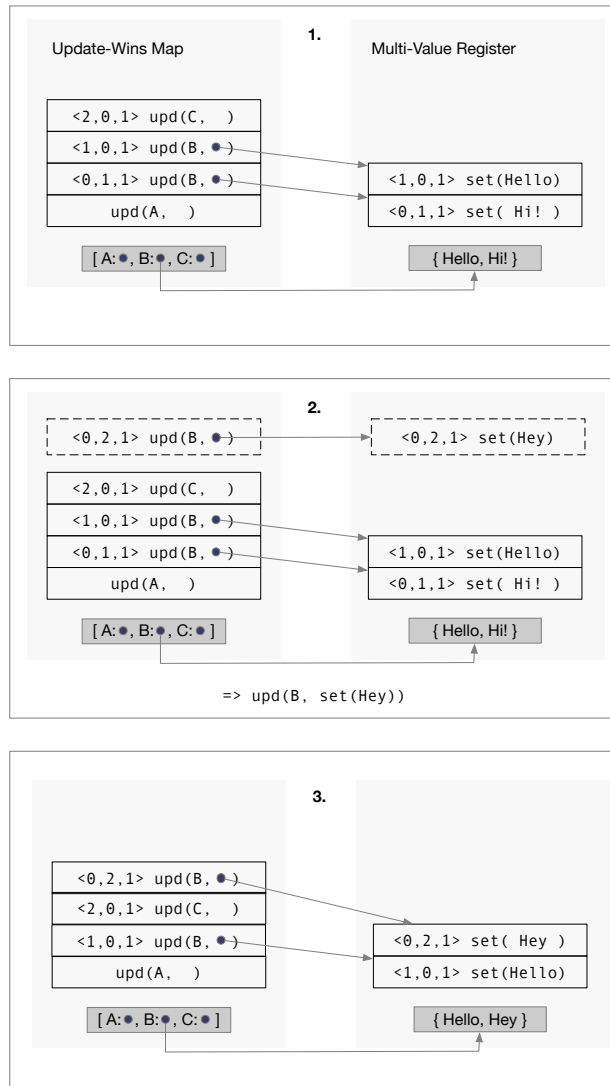


Figure 6.1: Three stages of the internal state of a hash-map with update-wins semantics containing nested Multi-Value registers: 1) initial state, 2) arrival of an update (upd) operation, and 3) final state after applying the operation.

follows others ( $\langle 0, 1, 1 \rangle, ..$ ). The update itself is applied to the hash map, making one of the existing update entries redundant, i.e., the one with vector clock  $\langle 0, 1, 1 \rangle$ , as it concerns the same key and has a non-concurrent timestamp. As the update operation itself is not redundant, its nested operation can be applied to the nested register. The `set(Hey)` is then applied to the nested register, making also one set operation redundant in the register, i.e., the one with vector clock  $\langle 0, 1, 1 \rangle$ . Note that there is another pair of concurrent operations in both the map and register that will not be made redundant, and thus are kept in the log. The third box shows the state and the log after applying `update(B, set(Hey))` resulting in the updated state `{Hello, Hey}`.

### 6.1.1.3 Maintaining Consistency of Children by Targeted Causal Resets

While applying redundancy checks on update operations ensures that the concurrency semantics of parents are upheld, it does not ensure that the concurrency semantics of *children* are upheld. In fact, the update mechanism ensures that redundancy relations are respected at each level of the CRDT, but these redundancy checks never cross hierarchical boundaries. This is expected as the redundancy rules are made for primitive, non-nested structures.

Consider a hash map with nested children; it is possible that a remove operation on the parent map is concurrent with some (but not all) operations on a child. The removal operation may make some, but not all, of the associated update operations redundant. The redundant updates will be removed at the parent level through the normal redundancy rules, but an additional mechanism would be needed to reflect the removal of the nested operations associated with the updates.

For this, we introduce a novel nested redundancy relation  $R_n$  that allows nested children to be reset to a particular logical timestamp (inclusive or exclusive of concurrent operations). With this relation, redundancy rules can be implemented to define hierarchical relations between log entries.

Figure 6.2 illustrates the use of our novel  $R_n$  relation in our running example of the update-wins map with nested Multi-Value registers. The first box (denoted by 1) shows the internal state and the PO-Log for the hash map, and the register associated with the key 'B' when a `delete(B)`

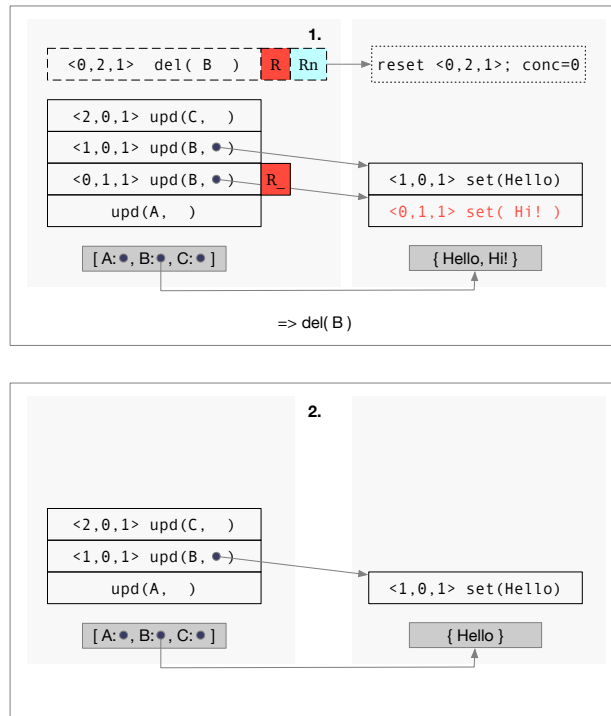


Figure 6.2: Example of a nested redundancy relation that selectively resets nested children, triggered by the deletion of a key. As the arriving delete (del) operation is concurrent with an update (upd) that arrived earlier, the nested child needs to be partially reset.

operation arrives. As this operation is concurrent with one of the earlier updates in the map, and the map follows update-wins semantics, the key itself cannot be removed. The entry with a preceding vector clock  $\langle 0,1,1 \rangle$ , however, will be marked redundant by the regular  $R_{\_}$  relation. At this point, the register associated with key B has partially redundant data, and as such needs to be updated to respect the remove operation. To this end, the introduced  $R_n$  relation can be used to reset all operations *in the nested register* that are previous to the delete operation. In the case of the example, the set of the value 'Hi!' (denoted in red in the figure) will be made redundant and removed from the register log. The second box shows the state and the log after applying the `delete(B)` operation in which all redundant operations are removed from the entire hierarchy, and the state of the register is updated to `{Hello}`.

### 6.1.2 Formalised Semantics for Extended Functionality

We now describe our approach as an extension of the formal model of a pure operation-based CRDTs framework (cf. Section 2.3). Algorithm 15 describes the distributed algorithm for our novel nested pure operation-based framework specifying the interaction between the RCB middleware and the framework, extending Algorithm 4 from Section 2.3.

**Notes on notation:** Recall that we used the  $i$  variable in Algorithm 4 to denote a particular replica. For example,  $s_i$  refers to the state of replica  $i$ . In the extended definition, we compound this with a list variable  $p$ , which denotes the path to the CRDT relative to its parent. The top-most data structure is denoted as *root*. For example,  $\{root, bob, favourite\_colours\}$  could be a path that refers to a `favourite_colours` object associated with the key 'bob' in a map. As such, we use  $s_{\{root, bob, favourite\_colours\}}$  to refer to the state of this object.

Algorithm 15 features the following new primitives for broadcasting and delivering nested operations:

- **broadcast\_nested $_{i,p}(o)$** : broadcasts nested operations ensuring that the operation will be delivered to all replicas (reliably and in causal order). In our design, a broadcast can only be triggered from the top-most data structure, as such  $p$  will always be *root*.

---

**Algorithm 15:** Distributed algorithm (for a replica  $i$ ) showing the interaction between the RCB middleware and the pure operation-based CRDT framework.

---

```

state:  $s_{i,p} := \emptyset$ 
state:  $children_{i,p}$ 
on  $operation_i(o)$  :
    |  $broadcast_{i,root}(o)$ 
on  $nested\_operation_i(p, o)$  :
    |  $broadcast\_nested_{i,root}(update(p, o))$ 
on  $deliver\_nested_{i,p}(t, update((child, \emptyset), o))$  :
    |  $deliver_{i,p}(t, update(child))$ 
    |  $deliver_{n,child}(t, o)$  if  $(t, update(child)) \mathcal{R}_{s_{i,p}}$ 
on  $deliver\_nested_{i,p}(t, update((child, p), o))$  if  $p \neq \emptyset$  :
    |  $deliver_{i,p}(t, update(child))$ 
    |  $deliver\_nested_{n,child}(t, update(p, o))$  if
      |  $(t, update(child)) \mathcal{R}_{s_{i,p}}$ 
on  $deliver_i(t, o)$  :
    |  $s_{i,p} := (s_{i,p} \setminus \{(t', o') \mid \forall (t', o') \in$ 
      |  $s_{i,p} \cdot (t', o') \mathbf{R}_{\perp}(t, o)\}) \cup \{(t, o) \mid (t, o) \mathcal{R}_{s_{i,p}}\}$ 
      |  $reset_{i,child}(t, 0) \mid \forall child \in children_{i,p} \cdot (child, 0) \mathbf{R}_n(t, o)$ 
      |  $reset_{i,child}(t, 1) \mid \forall child \in children_{i,p} \cdot (child, 1) \mathbf{R}_n(t, o)$ 
on  $stable_{i,p}(t)$  :
    |  $s_{i,p} := stabilize_{i,p}(t, s_{i,p})[(\perp, o)/(t, o)]$ 
    |  $stable_{i,child}(t) \mid \forall child \in children_{i,p}$ 
on  $reset_{i,p}(t, conc)$  :
    |  $s_{i,p} := s_{i,p} \setminus \{(t', o') \mid \forall (t', o') \in s_{i,p} \cdot ((t' \prec t) \vee (conc \neq 0 \wedge t' \parallel_c t))\}$ 
    |  $reset_{i,child}(t, conc) \mid \forall child \in children_{i,p}$ 
    
```

---

- **deliver\_nested** $_{i,p}(t, o)$ : called when an operation  $o$  is delivered (e.g. after it was previously broadcasted) on a replica  $i$  at path  $p$  with causal clock  $t$ .
- **nested\_operation** $_i(p, o)$ : called when a nested operation  $o$  needs to be applied at path  $p$ .

Recall from Section 6.1.1.2 that when an operation is applied to a nested child, at each level of the parent hierarchy, an *update* operation needs to be applied so that all redundancy rules can be checked. In the algorithm, the implementation of *nested\_operation* ensures that an operation is packaged in an *update* operation and broadcasted using *broadcast\_nested*. These broadcasted operations are received by the top-level data structure (root) using *deliver\_nested*. *deliver\_nested* will then try to deliver the operation to the child data structure specified by the path. At each level of the path, it will apply the *update* operation, check if the operation is not redundant, and if not, recursively descend into the hierarchy until the path only consists of one final child. It will then apply the actual operation to the last nested data structure using the non-nested *deliver*. Our approach extends the original *deliver* method with our novel nested redundancy relation: an implementation can use  $R_n$  to select what timestamps should become redundant for which nested children. Children are then (recursively) reset using the *reset* function, which takes a timestamp  $t$  and a variable *conc* that denotes whether the reset is exclusive (only entries that happened-before) or exclusive (including all concurrent entries).

### 6.1.3 Nested Pure Operation-Based Maps

In this section, we illustrate our framework by describing the design of two novel nested map CRDTs: an update-wins map (UW-Map, informally described in Section 6.1.1) and a remove-wins map (RW-Map).

Table 6.1 shows the semantics for the UW-Map in our pure operation-based framework. The design of the UW-Map CRDT is inspired by the add-wins Set CRDT [BAS17, BGB21], with some modifications to take care of its nested nature. The  $R$  relation for the UW-Map defines that **delete** operations will never be stored in the log (i.e., they are immediately redundant). The  $R_{\_}$  relation will make any existing operation in the log redundant if it happened before. This ensures that keys can be

deleted. Note that the  $R_{\_}$  relation also makes **update** operations with the same key that happened before be redundant. This makes the data structure a bit more efficient. Finally, the  $R_n$  relation for UW-Map defines that all nested operations that happened before any delete need to be recursively reset (i.e. removed). As this remove should be exclusive, i.e., no concurrent entries should be removed, we additionally encode that *conc* should be zero.

Table 6.1: Update-wins pure operation-based map, with support for nested CRDTs.

Framework	$(t, o) \mathbf{R} s$	$= \text{op}(o) = \text{delete}$
	$(t', o') \mathbf{R}_{\_} (t, o)$	$= t' \prec t \wedge \text{arg}(o) = \text{arg}(o')$
	$(child, conc) \mathbf{R}_n (t, o)$	$= conc = 0 \wedge \text{op}(o) = \text{delete} \wedge \text{arg}(o) = child$
	$\text{stabilize}(t, s)$	$= s$
User	$\text{update}(p, o)$	$= \text{nested\_operation} ([\text{op}=\text{update}, \text{arg}=[p, o]])$
	$\text{delete}(c)$	$= \text{operation}([\text{op}=\text{delete}, \text{arg}=e])$

Table 6.2: Remove-wins pure operation-based map, with support for nested CRDTs.

Framework	$(t, o) \mathbf{R} s$	$= \text{op}(o) = \text{update} \wedge (\exists (t', o') \in s \cdot \text{arg}(o) = \text{arg}(t') \wedge \text{op}(o') = \text{delete} \wedge t \parallel_c t')$
	$(t', o') \mathbf{R}_{\_} (t, o)$	$= t' \prec t \wedge \text{arg}(o) = \text{arg}(o') \wedge \text{op}(o) = \text{delete}$
	$(child, conc) \mathbf{R}_n (t, o)$	$= \text{op}(o) = \text{delete} \wedge \text{arg}(o) = child$
	$\text{stabilize}(t, s)$	$= s$
User	$\text{update}(p, o)$	$= \text{nested\_operation} ([\text{op}=\text{update}, \text{arg}=[p, o]])$
	$\text{delete}(c)$	$= \text{operation}([\text{op}=\text{delete}, \text{arg}=e])$

An alternative to update-wins is ensuring that delete operations are ordered after concurrent updates, leading to a map with remove-wins semantics. Table 6.2 shows the implementation of such a remove-wins map (RW-Map) in our framework. It is structured similarly to the AW-Map but has some additional complexity as the log needs to retain all delete operations until they are causally stable. The  $R_n$  relation encodes that

all previous or concurrent nested updates need to be removed (to ensure remove-wins semantics).

In this design of an RW-Map, in theory, `update` operations do not need to be stored in the log as these updates are stored in the nested children. However, only the last update operation for a particular child is kept (since previous update operations are removed from the log as they are redundant) As such, storing the `update` operations in the log can be useful to check if a particular child has a value, without having to query the nested children. When storing these entries poses a problem memory-wise, they can trivially be removed with no impact on the behaviour of the data type.

#### 6.1.4 Discussion

The implementation of the nested map CRDTs demonstrates that supporting nested structures can be tackled in a structured way. Our framework handles all logic related to nesting and update propagation, aiming to provide an easy-to-use interface. Additionally, hierarchical redundancy rules can be encoded using the  $R_n$  relation, ensuring that concurrency semantics are upheld at any level.

We believe that our approach simplifies the design of replicated nested CRDTs, and with it, we aim to reduce their implementation complexity. With the presented methodology, one can think of every CRDT with nesting support as a flat CRDT, which needs to support one additional operation, namely *update*. For example, a map is similar to a set of keys with an associated value. In a set, we can add and remove keys. Using some rules we can make the set add-wins or remove-wins, and with a bit of extra work, we can define how an *update* operation could be ordered against concurrent add and remove. This could be the core design of a Map. Our framework will make sure that every nested operation, e.g. a nested operation to a child of the map, is first represented as an *update* operation for the parent CRDT. The parent CRDT (e.g. the map) does not need to know anything about the nested content of this update, it is simply trying to make sure that this update will be properly ordered between the additions and removals of keys. This alone, however, is not enough to ensure convergence, i.e. that the algorithm is correct. Depending on the arrival order of an update in combination with other concurrent operations, the associated nested operation may have been applied to some

replicas and not to others. To ensure that the nested state converges, the algorithm sometimes might need to apply some cleanup procedures, which is precisely where the nested redundancy relation comes into play. In Section 6.3.1 we formally prove that this is the case for our approach and our implemented designs.

## 6.2 Implementation

In this section, we describe the implementation of our novel nested pure operation-based approach in Flec. We focus on the extensions to Flec required to support nested pure operation-based CRDTs. We expose our work as extensions of the operation-based CRDT API, as described in Table 3.5 from Section 3.3.2. Section 6.2 summarises the constructs that were added to support our implementation.

### 6.2.1 Implementing Nested CRDTs in Flec

We now illustrate the extended Flec by means of the RW-Map CRDT described in Table 6.2. Listing 6.1 and Listing 6.2 show the core of the implementation of RW-Map CRDT in Flec. Lines 4 to 8 in Listing 6.1 define the CRDT constructor, which is used to initialise the `values` property that contains all nested children. Additionally, an initialiser can be specified that sets the initial (start) value for children. For example, if a map with a nested AW-Set is needed, the initializer will initialize a new AW-Set CRDT. Lines 14-16 in Listing 6.1 show the update function which can be used to apply nested operations on children (by CRDT client code). Any operation on a child is indicated by specifying a particular path, and the update to be applied. Using `performNestedOp` this operation will be propagated to the child and all replicas. The actual semantics can be seen in Listing 6.2 which shows the implementation of the redundancy relations and children referencing.

Lines 20 to 22 in Listing 6.2 show the implementation of the `resolveChild` method which allows Flec to reference children, stored in the `values` property. The rest of the listing shows how the RW-Map implements redundancy relations to achieve remove-wins semantics: the RW-Map provides an implementation for `isPrecedingOperationRedundant` to implement the  $R_{\_}$  relation: any operation in the log is redundant if

Table 6.3: Nesting Operation-Based CRDT Interface in Flec.

<b>Nesting Extensions (class: PureOpCRDT)</b>	
<b>Event Handling Methods (For CRDT Implementors)</b>	
<code>doesChildNeedReset</code>	Encodes the $R_n$ binary relation (i.e., from what timestamps do children need a partial reset).
<b>Command Interface (For CRDT Implementors)</b>	
<code>performNestedOp</code>	performs a nested operation and broadcasts it to other replicas.
<code>setChildInitialiser</code>	Method that will be used to initialise new children, using child-specific constructs (e.g. if you want children to be AW-Sets, the initialiser will return a new AW-Set).
<code>addChild</code>	Register a CRDT as a child to a parent, for a particular key.
<b>Command Interface (For Framework Extenders)</b>	
<code>resolveChild</code>	Used internally to resolve nested CRDTs. Can be used to override the default internal child bookkeeping and change how the the framework resolves child CRDTs (this will disable add-Child).

Listing 6.1: The implementation of an RW-Map in Flec, using the described extensions (A).

```
1 export class RRWMap extends PureOpCRDT<MapOps> {
2   values: Map<string, NestedCRDT>;
3
4   constructor(initializer: () => NestedCRDT) {
5     super();
6     this.values = new Map();
7
8     this.setChildInitialiser(initializer);
9   }
10  ...
11  // User functions
12  ...
13
14  public update(path, ...args) {
15    this.performNestedOp("update", path, args);
16  }
17 }
18
```

Listing 6.2: The implementation of an RW-Map in Flec, using the described extensions (B).

```
1 protected isPrecedingOperationRedundant(existing: MapEntry,
2     arriving: MapEntry, isRedundant: boolean) {
3     return arriving.isDelete() && existing.hasSameArgAs(arriving);
4 }
5
6 protected isArrivingOperationRedundant(arriving: MapEntry) {
7     const concurrentDeletes = this.getConcurrentEntries(arriving).
8         filter(e => e.entry.isDelete() && e.entry.hasSameArgAs(
9         arriving));
10
11     return concurrentDeletes.length > 1;
12 }
13
14 protected doesChildNeedReset(child, arriving: MapEntry) {
15     return {
16         condition      : arriving.isDelete() && arriving.args[0]
17         == child,
18         reset_concurrent: true
19     };
20 }
21
22 // Resolve child CRDTs
23 protected resolveChild(name: string) {
24     return this.values.get(name);
25 }
```

it has happened before a newly arriving operation, and if they are acting upon the same child. It also implements `isArrivingOperationRedundant` to define the  $R$  relation: any arriving update is not applied if a concurrent delete is stored in the log. Finally, by providing an implementation for `doesChildNeedReset` we specify that when a delete arrives for a particular child, the child will be reset. The `reset_concurrent` flag is set to true to indicate that even concurrent updates to the child should become redundant.

## 6.3 Validation

To validate our nested pure-operation based CRDT approach, we conduct three experiments. First, we verify the correctness of our proposed framework and nested pure op-based maps. Secondly, we implement the concepts in a real programming framework and finally, we compare it to another framework featuring similar concepts.

### 6.3.1 Verification with VeriFx

In order to verify our approach, we have re-implemented the core of our nested pure operation-based CRDTs in VeriFx [DPFGB23]. VeriFx is a programming language for replicated data types with automated proof capabilities that allow users to implement replicated data types in a high-level language and express correctness properties that are verified automatically. VeriFx internally uses an SMT theorem prover to search for counterexamples for each property that needs to be upheld. It also enables the transpilation of the data types to mainstream languages (e.g. Scala and JavaScript).

Correctness means that CRDTs built with the framework exhibit the *strong convergence* property. As explained in Section 2.1.2.2, this requires that replicas need to have received the same operations to be in the same state (regardless of the order in which the operations have been received). Shapiro et al. showed in [SPBZ11a] that operation-based CRDTs guarantee strong convergence if all concurrent operations commute. In our case, this implies checking the effects of all redundancy relations. Proving the correctness is, however, slightly trickier in our case, as we are dealing with a recursive design. SMT solvers, such as Z3 used by VeriFx, do not deal well with recursive and nested data structures, as they might not be able to find a solution in a finite time. To verify our approach, we thus combine VeriFx proofs with structural induction, which limits the recursion depth needed to verify our design:

- Base case: we implemented a 'perfect' resettable pure operation-based CRDT in VeriFX that can model both a flat CRDT or a CRDT containing children. The CRDT logs all operations in a single flattened log (e.g., one log for all potentially nested structures). Items in the log can be reset by a parent when requested. No redun-

Listing 6.3: Convergence update-update.

```

1 proof FUWMap_update_update_converges {
2   forall(map: FUWMap, k1:String, k2: String, t1: VersionVector, t2
3     : VersionVector, o1: SimpleOp, o2: SimpleOp) {
4     ( t1.concurrent(t2)  && map.children.contains(k1) && map.
5     children.contains(k2) &&
6       map.polog.forall((e:TaggedOp[FMapOp])=>
7     ((e.t.before(t1) || e.t.concurrent(t1))
8       && (e.t.before(t2) || e.t.concurrent(t2)
9     )))) => (
10
11     map.update(t1, k1, o1).update(t2, k2, o2)
12     ==
13     map.update(t2, k2, o2).update(t1, k1, o1)
14   )
15 }
16 }

```

dancy rules are applied. This design ensures that we can represent a 'correct' nested structure (in terms of SMT assumptions) without needing a recursive model. We use a VeriF<sub>x</sub> proof to ensure convergence of this 'perfect' CRDT.

- Induction step: a particular nested CRDT can be implemented on top of our VeriF<sub>x</sub> implementation and set to use perfect nestable CRDTs as children. With this approach, VeriF<sub>x</sub> can then be used to prove that our approach is correct for one level of nesting, for all pairs of operations.

By combining the base case and induction step, we prove using structural induction that our framework remains correct for any nestable structure.

As an example, Listings 6.3, 6.4, and 6.5 show the VeriF<sub>x</sub> proof logic that was used to check the behaviour of concurrent operations on an update-wins map implemented with our framework. We define that any pair of correct operations that are concurrent and applied to a correct state should commute. The operations and state are correct if the operations (causally) follow or are concurrent with all other operations that

Listing 6.4: Convergence update-delete.

```

1 proof FUWMap_update_delete_converges {
2   forall(map: FUWMap, k1:String, k2: String, t1: VersionVector, t2
3     : VersionVector, o1: SimpleOp) {
4     (t1.concurrent(t2) && map.children.contains(k1) &&
5     map.polog.forall((e:TaggedOp[FMapOp])=>((e.t.before(t1) || e.t
6     .concurrent(t1)) && (e.t.before(t2) || e.t.concurrent(t2))))
7     =>: (
8       map.update(t1, k1, o1).delete(t2, k2)
9       ==
10      map.delete(t2, k2).update(t1, k1, o1)
11    )
12  }
13 }

```

Listing 6.5: Convergence delete-delete.

```

1 proof FUWMap_delete_delete_converges {
2   forall(map: FUWMap, k1:String, k2: String, t1: VersionVector, t2
3     : VersionVector, o1: SimpleOp, o2: SimpleOp) {
4     (t1.concurrent(t2) && map.children.contains(k1) && map.children.
5     contains(k2) && map.polog.forall((e:TaggedOp[FMapOp])=>((e.t.
6     before(t1) || e.t.concurrent(t1))
7     && (e.t.before(t2) || e.t.concurrent(t2))))
8     =>: {
9       map.delete(t1, k1).delete(t2, k2) == map.delete(t2, k2).
10      delete(t1, k1)
11    }
12  }
13 }

```

were applied previously to the state (e.g. everything in the log). For this definition, we assume the usage of RCB (which is the case with the pure operation-based CRDT framework), so that we know that everything in the log must be concurrent or happened-before. In other words, the logic encodes the correctness properties that should always hold in our framework, i.e. that if all operations on the map commute and the nested operations are applied to correct CRDTs (in our case, all nested operations are applied to a 'perfect' CRDT), that the map is correct.

We use VeriF<sub>x</sub> to verify these properties hold given the implemented designs. Internally, the VeriF<sub>x</sub> SMT engine will look for valid solutions that satisfy the negation of our definitions, it will search for any case where the correctness properties are violated. Since no counterexamples (valid solutions for the negation of properties) were found after exhausting all search options, we can then constitute that our framework model is valid according to the correctness properties.

Using this approach, we have verified our map designs, validating both the concurrency semantics of our proposed CRDTs and proving that our novel framework functions correctly. The benefit of our verification approach is that to validate the correctness of any nestable CRDT (built on our framework), one only needs to encode proofs for the operations on a flat level. All needed nesting aspects of the proof will automatically be inherited from our VeriF<sub>x</sub> implementation.

### 6.3.2 Portfolio of Nested CRDTs in Flec

To show the flexibility and applicability of our approach, we have implemented several commonly used data structures as novel nested pure operation-based CRDTs in Flec, summarised in Table 6.4. As shown in the previous section, we have map implementations with update-wins and remove-wins semantics. Maps form the basis for many other data structures and thus are essential to any replication framework. They have been verified using their VeriF<sub>x</sub>-based implementations and have been used in more complex data structures since.

We have implemented two other maps: one modified map (based on the remove-wins map) that optimises some structures to have better memory resource usage, and another map where keys are managed by an add-wins set. Finally, we have a delete-wins list that can be used to store values in

Table 6.4: Implemented nested CRDT types.

CRDT	Semantics
UW-Map	Update-wins map where values can be CRDTs. Update win from concurrent deletes.
RW-Map	Remove-wins map where values can be CRDTs. Deletes win from concurrent updates.
RW-Map (mod)	Modular version of the remove-wins map that allows more efficient memory usage.
AW-Map	A variant of the update-wins Map where keys are managed by an add-wins set.
AW-Set	An add-wins set where values can be CRDTs.
DW-List	A delete-wins linked list where elements can be CRDTs.
ImmutableCRDT	A map with immutable keys, which behaves similarly to structs in C.

sequential order. Similarly to other sequential replicated structures such as RGAs [RJKL11], a linked list is used internally.

### 6.3.3 Use-Case: A Mixed CRDT-Based Distributed Filesystem

To validate our approach in a real-world application scenario, we implemented a distributed file system based on the work of [YYRB21] in Flec. This application is also used later in Section 6.3.4 to compare our approach to state-of-art.

As mentioned in Section 3.3.2, Flec comes with a portfolio of generic CRDTs, mostly pure operation-based CRDTs. While our extensions to Flec are focused on pure operation-based CRDTs, part of the added nesting support can also be used in conjunction with general non-pure operation-based CRDTs to develop real-world applications.

When composing traditional CRDTs, operations on a (parent) root node typically trigger several operations that will be applied to internal (nested) CRDTs. For a single operation, these sub-operations need to be applied atomically, they cannot be viewed as independent and should not automatically replicate to nested children of replicated CRDTs. This

is in contrast with our main approach where an update is applied via a particular sub-path. To ensure compatibility with this approach in the framework, nested children can detect the context in which operations are applied. If a nested CRDT has a parent, and an operation is applied directly from that parent (and not via a nested update), the operation will not be broadcasted to other replicas. Instead, it is assumed that the (top-)parent operation will be broadcasted, resulting in the same nested update path on other replicas.

We now discuss the overall data structures and operations of the distributed file system.

Appendix A shows the core of the implementation. It has been modified to hide some minor boilerplate code, type definitions, and a lot of operation handling code, but it contains the essentials. Listing A.3 shows the main body of the `DistributedFS` class, which implements the core functionality of the CRDT. By extending the `OpCRDT` class it automatically inherits all the distribution and CRDT functionality from `Flec` (along with our extensions).

Lines 5-21 define the required data structures for the distributed file system that keep track of metadata for files, groups and users. To this end, we define three maps, and each map on its own contains records (in the form of `ImmutableCRDT`) containing other CRDTs for storing the metadata of particular files, groups and users. For example, the `files` data structure is defined using an `RW-Map` and contains filesystem metadata related to access rights, ownership, and data content. The data types we use for the registers (`AccessRightF`, `UserID`, ...) are basic types constructed from primitive types such as numbers or strings and can be stored directly in the registers. `AccessRightF` is a numerical value that we index as a bit-vector to store our permission flags (similar to POSIX systems). We provide an additional TypeScript class, `AccessRight`, that provides a high-level abstraction to this bit-vector, but concretely we store numerical values in the CRDT register. Lines 24-28 define the `onLoaded` method which associates the aforementioned three maps with their parent CRDT. In line 30, the `setHandler` method defines all operation handlers which implement the semantics of the CRDT.

Listing A.4 shows the implementation of the `CreateFile` operation in more detail, and Listing A.5 shows code that exposes some of the CRDT API to the local user, for performing some basic actions which are used

by the `test` method in Listing A.6 to show local usage of the file system functionality. Flec will ensure that all operations are properly replicated and distributed. In general, most of the code is similar to that of sequential data structures, and the API is not much more complex. This is in line with the goal of our framework: an easy-to-use interface for building CRDTs where developers can immediately benefit from a middleware that does all the heavy lifting.

### 6.3.4 Evaluation of Network Traffic in Comparison With Automerge

To compare our approach with state of the art, we implemented the same distributed filesystem in Automerge v1.0.1 [KB17] and evaluated the differences in network traffic between our Flec implementation and the Automerge one.

Note that it is not possible to select the individual concurrency semantics for nested objects with Automerge, as is possible with our extension to Flec. As such, the Automerge implementation has a slight difference in concurrency semantics when compared to the original design [YYRB21] and our implementation. For example, while the distributed filesystem (DFS) specification describes update-wins concurrency semantics for the user list, the Automerge implementation uses remove-wins concurrency semantics. Functionality-wise, it has the same features. In fact, in our implementations, both the Automerge and Flec versions have the same API.

As explained in Section 2.4.1, Automerge is network agnostic and does not provide a network layer; instead it provides an API that allows you to query (Automerge) documents for changes. If any changes exist, you can propagate these over any networking channel that your application depends on. On the receiving end, you can insert these changes back into Automerge, which can merge the received information in the local state. Automerge itself uses a state-based approach, where only the required changes (deltas) are propagated instead of the full state, to conserve network bandwidth.

For the experiments, we used a virtual network for both Automerge and Flec, which allows us to reproduce benchmarks and results with little non-determinism. We set up a system with 5 nodes (ad-hoc, peer-to-peer), and issue a thousand operations per experiment.

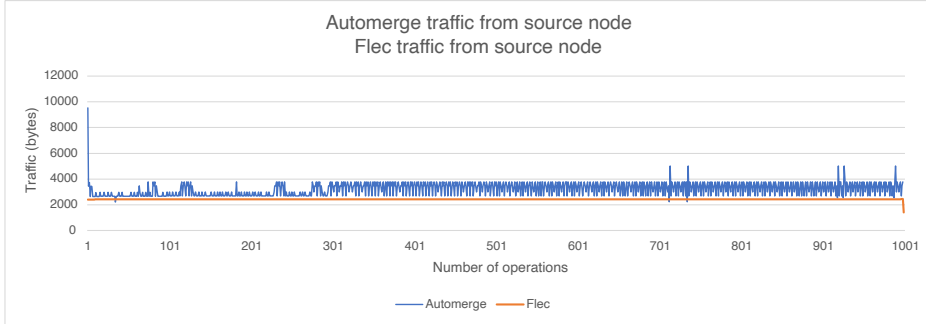


Figure 6.3: Network traffic (in bytes/op) originating from the source node for both Automerge and Flec. In every operation, a file is created and written.

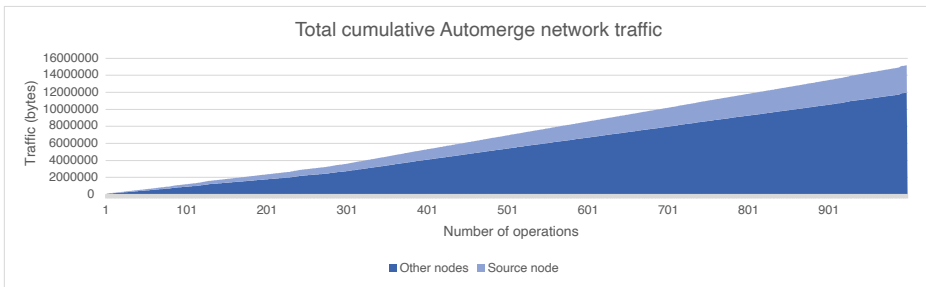


Figure 6.4: Total cumulative networking traffic (in bytes/op) from all nodes for Automerge. In every operation, a file is created and written.

#### 6.3.4.1 Experiment A: File Creation and Writing

For the first benchmark, each operation exists out of file creation and file modification. We applied these operations a thousand times to a deployed distributed file system, once using the Flec implementation and once with the Automerge implementation.

Figure 6.3 shows the network traffic originating from the source node (the node where the operations are applied), for both implementations. As both our approach and Automerge share the essential updates, the results are fairly stable and linear. Automerge will always send small updates containing the state delta (which means the newly modified file) and our extension to Flec sends the operations itself. While Automerge uses a binary representation for the update payload, the payload itself is still heavier than the non-optimized JSON payload used in Flec.

The visualisation hides some essential information, however. Automerge uses an additional protocol that allows replicas to propagate updates among each other. This means that not only the source node will share information, but also other nodes that received the new updates if they believe that other replicas may be missing information. Figure 6.4 highlights the additional traffic, showing that it makes up a significant portion of the total network traffic. In Flec updates are only sent directly from a source node to a destination node, and as such, there is no additional network usage.

#### 6.3.4.2 Experiment B: User, Group, and File Creation, and Configuration

For the second experiment, in each operation, we create a new user, and a new user group, add the user to the new group, create a new file (with the new user as owner), and write to this file. This extra complexity leads to some interesting results. As seen in Figure 6.5 the Automerge measurements stop at around  $\sim 100$  operations. This is because the additional gossip traffic starts growing exponentially (see Figure 6.7) and causes the entire system to halt. We are not exactly certain what causes this problem, but we did not observe this issue with the previous experiment, only when we applied more complex operations. We believe that this is not correct behaviour from Automerge, but we were not able to identify the root cause of the bug. The behaviour is consistent and reappears with

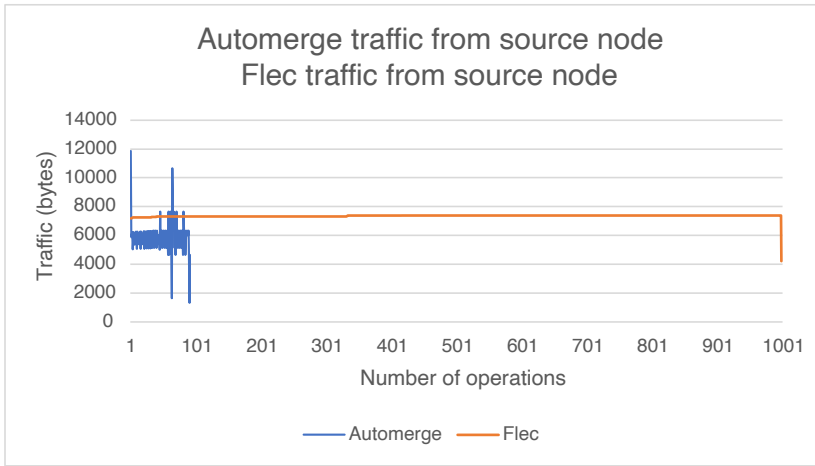


Figure 6.5: Network traffic (in bytes/op) originating from the source node for both Automerge and Flec. Every operation creates a new user, a new group, and a new file. The user is added to the group, and the file is created with the new user as the owner. Finally, the file is written.

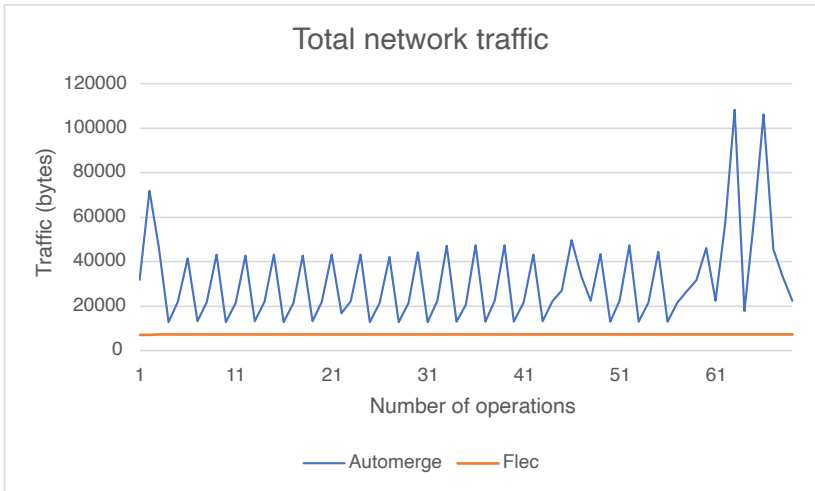


Figure 6.6: Total network traffic (in bytes/op) for both Automerge and Flec. Every operation creates a new user, a new group, and a new file. The user is added to the group, and the file is created with the new user as the owner. Finally, the file is written.

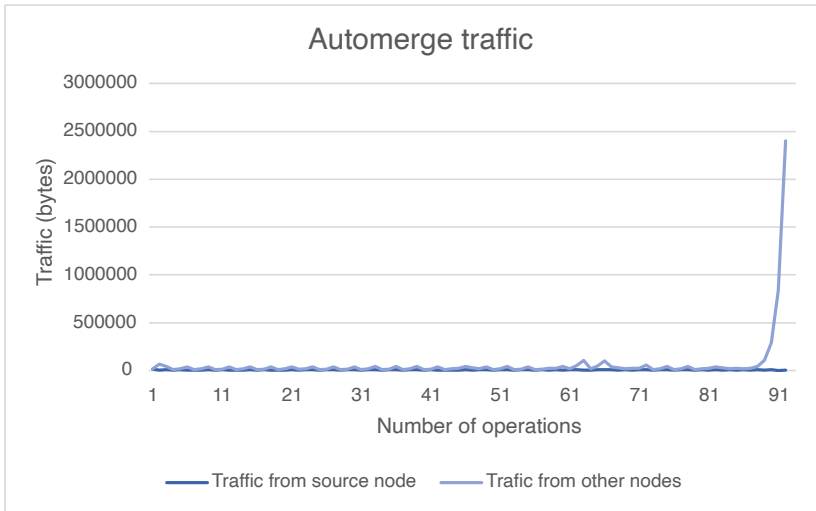


Figure 6.7: Total network traffic for Automerge for the previous experiment, highlighting an issue with exponential growth after a certain number of operations.

each run. To be able to evaluate this example anyway, we will only focus on the initial measurements before the exponential explosion. Based on Figure 6.5 we can see that Automerge has a lower network overhead on the source node when compared to Flec. When looking at the total traffic, however (Figure 6.6), we can see that Automerge still utilizes more bandwidth. The reason for this is that as we are sending many operations, other replicas start propagating updates as well, resulting in the source node itself sending fewer updates (as it is relieved from work).

### 6.3.4.3 Conclusion

With this experimental evaluation, we showed that our approach is comparable to state-of-the-art CRDT frameworks, even though Flec and our extensions have not yet been optimised for non-experimental use. While additional optimisations can be applied to the pure operation-based CRDT framework and our nested framework extension, these results are promising and show that our approach is viable in real-world scenarios.

## 6.4 Notes on Related Work

As explained in Section 2.4.3, the composition of replicated structures is possible in a few frameworks like Automerge [KB17] and Lasp [MVR15]. While Automerge allows programmers to arbitrarily nest linked lists and maps in a document, it doesn't allow for much flexibility regarding the actual merging semantics, as shown by the implementation of the distributed filesystem in Section 6.3.3. Lasp supports functional transformations over existing CRDTs provided in the language, which allows a composition to some extent. However, when the portfolio of CRDTs falls short in those frameworks, developers need to design the desired nested data structure from scratch.

Weidner et al. [WMM20] introduce techniques for creating novel CRDTs based on existing (de-composed) CRDTs *with a static structure*. In our approach, nested data structures can change dynamically during runtime, using maps, lists, and sets.

Preguiça et al. explain in [Pre18] several possible nesting semantics for operation-based CRDTs and discuss the need for CRDTs to support partial resets. Our approach follows this idea and introduces a recursive reset mechanism to the pure operations-based CRDT framework. This allows for systematically using any CRDT as nested values for structured CRDTs without modifying their semantics.

## 6.5 Conclusion

This chapter explored a structured approach for designing nested CRDTs based on the pure operation-based CRDT framework. We propose a novel framework for building nested pure operation-based CRDTs and show how several common nested data structures can be designed and modelled in the framework. We validate our approach by extending our pure operation-based framework in Flec to include support for nested pure operation-based CRDTs and implement a portfolio of commonly nested data structures. Concretely, this chapter introduces the following contributions:

- A general approach for designing and implementing nested CRDTs, building on the work of pure operation-based CRDTs.

- A full-fledged implementation of our approach in Flec, including a portfolio of existing and novel pure operation-based CRDTs.
- A validation of the correctness of our nested pure operation-based framework and a portfolio of CRDTs built on this framework through VeriF<sub>x</sub>.
- A performance evaluation of an application scenario showing that our approach has reduced network usage when compared to Automerge [KB17].

While our work focuses on the pure operation-based CRDT framework, we have shown through the distributed file-system example, which combines classic operation-based CRDTs with pure operation-based CRDT structures, that our work is general enough to be used in various other settings. We believe it should be possible to emulate our approach as a state-based design, although an efficient design may be hard to achieve.

To conclude, we presented a novel approach to nested pure operation-based CRDT framework, which allows for the systematic implementation of structured CRDTs. We believe that our work forms an important contribution that will help design and create complex local-first and geo-distributed applications.



# Chapter 7

## Conclusion

This dissertation introduced a CRDT framework for systematically handling important concerns regarding distribution and replication, memory management, and nesting and composition of data structures. This chapter concludes this dissertation by summarising our key contributions and discussing the limitations of our approach.

### 7.1 Restating Our Approach

In this research, we argue the need for a systematic approach to ease the design and implementation of CRDTs. We build Flec, an extensible CRDT framework implementation to support our exploration of systematic designs. Flec provides an open implementation that allows it to be used as a laboratory for experimenting with CRDTs.

This thesis studied what concerns can be addressed directly in a framework and what aspects need to be exposed to the data type itself. Concretely, we explore and design solutions for the following problems:

**Lack of support for dynamic networks.** In Chapter 4, we discussed how most CRDT specifications are defined for fixed networks, i.e., networks where the number of participants is fixed. We argue that this assumption greatly limits the potential of CRDTs in dynamic environments where new participants can join freely. This is, for example, an important aspect of local-first software, where users can collaborate without requiring a centralised server for coordination

among devices, and new devices can participate at any moment in time.

We propose a mechanism to enable the use of CRDTs in dynamic networks. The mechanism uses a protocol through which new nodes can coordinate with the existing nodes in a network to obtain a correct state. Our approach is not tied down to any particular CRDT design and provides a systematic way to deal with dynamic networks.

**Slow Metadata Removal.** In Chapter 4, we also discuss memory management issues. We show how existing approaches for metadata removal depend on updates from other nodes to determine causal stability. This is problematic, as such updates are not always guaranteed, especially in dynamic networks where nodes are often offline.

We introduce an approach to eagerly determine causal stability by piggybacking on reliable delivery guarantees. The approach only requires modifications to the reliable delivery mechanisms and is not specific to any CRDT data type.

**Unreactive CRDTs.** Chapter 5 reviewed the effects of causal delivery on CRDT designs and showed how it may lead to CRDTs with reduced reactivity; i.e., they might delay operations needlessly.

We introduce an approach that improves the reactivity of CRDT designs that rely on causal delivery by the reification of buffered operations. We apply this approach to the pure operation-based CRDT framework, where the buffer is exposed through a new redundancy relation. This allows existing CRDT designs to be extended without their core semantics needing to be modified.

**Limited Support for Complex Data-Structures.** In Chapter 6, we review the composition of CRDT types. Typically, end-users are limited to the usage of restrictive types and are unable to nest arbitrary CRDTs with custom convergence semantics.

We introduce a framework for the dynamic nesting and composition of CRDTs, and apply our approach to the pure operation-based CRDT framework. Our approach allows for the systematic implementation of new nested CRDTs and the use of existing CRDTs within nested structures without any modifications.

Overall, we show that our approach enables the implementation of solutions that can be systematically applied to existing CRDT designs and frameworks.

## 7.2 Contributions

In this section, we restate and summarise our contributions:

**Eager Causal Stability in CRDTs.** A novel mechanism to eagerly determine causal stability, improving causal metadata removal. The approach allows for a faster metadata cleanup process and allows systems to have a lower memory consumption. We implemented our approach in Flec and evaluated the memory management techniques through several experiments. We show that our approach improves the metadata removal rate and optimises the efficiency of CRDTs.

**A Join Model for CRDTs in Dynamic Networks.** A join model to support dynamic networks, where new nodes can join at any moment. Our approach ensures new nodes can acquire a correct replication state, allowing them to participate effectively in the replicated system. We describe how our model can be integrated into systems without requiring modifications to CRDT designs and validate our approach by implementing it in Flec.

**Improved Reactivity for CRDTs.** We proposed an approach to improve the reactivity of frameworks relying on causal ordering through the reification of the buffered operations. We implemented our design in Flec, along with reactive versions of the Add-Wins and Remove-Wins sets. We compared normal and reactive sets in our implementation and showed that reactive CRDTs can achieve higher throughput than non-reactive versions in a system experiencing delays.

**Nestable Pure Operation-Based CRDTs.** We introduced nested pure operation-based CRDTs and showed how several common nested data structures can be designed and modelled in the framework. We validated our approach by extending our pure operation-based framework in Flec to include support for nested pure operation-based CRDTs and implement a portfolio of commonly nested data

structures. Through an evaluation we show that the performance of our approach is comparable to Automerge, a state-of-the-art CRDT library.

### 7.2.1 Technical Contributions

Throughout our dissertation, we use Flec, our open CRDT implementation framework for TypeScript. Flec provides the following features:

**TSAT:** TSAT provides the Ambient-Oriented Programming Paradigm (AMOP) to Flec. It enables the implementation of concurrent and distributed programming.

**RCB Middleware:** Flec has built-in support for an extensible Reliable Causal Broadcasting middleware. Through an MOP, the behaviour of the middleware can be extended or modified.

**CRDT Framework:** Flec provides an extensive and open CRDT framework for implementing and using CRDTs. It handles all replication and distribution aspects through the TSAT and RCB components. The framework can be extended through an extensive MOP, as demonstrated by the implementation sections of the various chapters of this dissertation. While we provide constructs to build state-based and operation-based CRDTs, our largest contribution is our extended pure operation-based CRDT layer.

**Portfolio of CRDTs:** We provide a portfolio of lists, sets, counters, registers, maps, and other common data structures, implemented as CRDTs on top of Flec. A subset of the implementations are based on existing approaches, extended to support the extensions from our work, while others are novel designs.

## 7.3 Discussion

In this section, we discuss potential limitations to our contributions.

### 7.3.1 Dynamic Join Model

In Chapter 4, we introduce the dynamic join model, which enables the use of CRDTs in networks where nodes can dynamically join. Currently,

we assume that all failures are transient and that partial failures will eventually be resolved. As a result, we do not model nodes dynamically leaving a network.

To handle the leaving of nodes, we foresee that a new synchronisation protocol will be required to ensure agreement on what nodes participate in a network. However, the exact method will depend on the possible tradeoffs.

For example, in IoT networks, additional explicit synchronisation steps might be too expensive. A potential solution in this case would be to mark nodes as `dead` in the vector clocks. When these clocks are eventually propagated to other nodes and reach causal stability, additional clean-up processes might be initiated. In geo-replicated systems, where networks have a large bandwidth, occasional coordination might not be expensive. As such, leaves could be handled through explicit coordination, e.g., with locks.

### 7.3.2 Eager Causal Stability

Our approach for Eager Causal Stability, as defined in Chapter 4, relies on extending the reliable delivery mechanisms of the delivery middleware. Currently, we include the full logical clocks in delivery acknowledgements. When a network grows, and consequently the number of nodes, the overhead of including such clocks will also grow.

To optimise resource usage, we foresee that the node sending the acknowledgement could attach an optimised version of the clock [ASB14]. For example, a subset of the logical clock, which can be determined by calculating a delta between the local clock and the operation clock, could be used. This would limit the size of the attached clock to the difference in clock values between nodes.

### 7.3.3 Nested Pure Operation-Based CRDTs

Our approach to Nested Pure Operation-Based CRDTs does not allow using CRDTs as keys to nested structures. We decided not to implement support for this behaviour as there are different interpretations of how key equality should be designed. It is possible to emulate this functionality by creating a structure that tracks both keys and values separately as lists and then using specialised query functions to associate the keys with

values. While this approach is highly inefficient, it could provide a basis for more efficient and native implementations in the future.

## 7.4 Future Work

In this section, we discuss potential extensions and future directions of work.

### 7.4.1 Reactive CRDTs

In Chapter 5, we extend the pure operation-based CRDT framework and introduce the  $R_\beta$  relation that tackles the redundancy of operations in the main log by evaluating buffered operations.

In future work, we would like to explore if operations from the buffer could be made redundant before all their causal dependencies have been delivered. This would bring an extra improvement in the reactivity of the CRDTs, but implementing this can lead to subtle problems if not done carefully. Unlike with the PO-Log, items do not arrive in causal order. This means that extra causal bookkeeping may be needed to track operations - possibly undoing the extra benefit.

### 7.4.2 Automatic Generation of Redundancy Relations

In this dissertation, we introduced several extensions to the pure operation-based CRDT framework through novel redundancy relations. We observed that datatype semantics were encoded several times in different ways in the specification of CRDT designs.

In future work, we would like to investigate the automatic generation of redundancy relations based on annotated sequential implementations. For example, we could annotate a sequential set implementation to specify that concurrent removes will always be ordered after adds when used in replicated settings, resulting in remove-wins concurrency semantics. The compiler could then use these annotations to automatically deduce redundancy relations, causal compaction rules, and query functions. This would ensure consistent behaviour between all rules and allow our framework to apply optimisations where possible.

### 7.4.3 Distributed Garbage Collection through Consensus

In Chapters 4 and 5, we explored various techniques for eagerly compacting our data structures and removing unneeded metadata. We generally adopted a conservative approach, aiming to minimise network overhead. For future work, we propose exploring a system that performs background synchronisation to enable the active removal of redundant metadata, allowing developers to define a trade-off between network and memory usage.

In our current approach, Flec relies on background *ping* messages to determine node availability. We believe this approach could be expanded by implementing a consensus algorithm, such as Paxos [Lam98], to synchronise the state of nodes immediately. Each message issued by the consensus algorithm could serve a dual purpose, both as a check for node availability and as part of the synchronisation process. We envision that the rate at which the consensus algorithm is executed could be adapted to optimise network communication usage.

### 7.4.4 Multi-Log Pure Operation-Based CRDTs

Currently, pure operation-based CRDTs use a PO-Log to track operations. Theoretically, this log can be viewed as a CRDT itself. It might be interesting to explore designs where the log is reified as a CRDT and experiment to see if it can be structured differently.

In particular, we believe it is worth investigating the possibility of combining several PO-Logs into a single CRDT to create multi-log pure operation-based CRDTs. We envision CRDTs where the operation log consists of several CRDT PO-Log segments. Operations could be tagged with metadata, which determines the destination segment. Segments could be dropped or moved to other replicas, depending on the situation.

As a concrete use case for multi-log pure operation-based CRDTs, consider an example where messages could be tagged with authentication information. When user permissions are revoked and authentication metadata modified, segments could be dropped to reflect the change. It would also be possible to scope redundancy relations based on authentication levels.

### 7.4.5 Smart Concurrency Semantics Through NLP

CRDT-based approaches are appealing for collaborative text editing systems as they incorporate strategies to ensure concurrent updates cannot conflict. However, these strategies often result in unexpected behaviour from the end-user's perspective, e.g. they may lead to duplicate words and grammatically incorrectly merged sentences. Future work could investigate the use of deterministic natural language processing (NLP) algorithms to improve the concurrency semantics of collaborative text editing systems that rely on CRDTs, with the aim of providing a better end-user experience [BDPGB23]. A CRDT could use NLP hints to steer the process of merging concurrent/conflicting operations when editing a document.

When there is no clear merging approach based on the NLP output, the framework should fall back to standard concurrency semantics. A systematic design for this could be implemented following techniques like those proposed for multi-log CRDT.

### 7.4.6 Comparison of Flec to Mainstream Frameworks

We aim to conduct a comprehensive comparison of Flec to mainstream frameworks in more realistic workloads. To this end, we will use the Yahoo! Cloud Serving Benchmark (YCSB) [CST<sup>+</sup>10] benchmark platform which is widely used for evaluating databases and offers a standardised framework for applying different workloads.

We have implemented a Java-based Flec backend for YCSB, which enables communication with Flec instances (either local or remote) running a newly developed key-value database service. This service is based on our nested pure operation-based CRDT framework and utilises update-wins maps CRDTs for the database tables and records, and last-writer-wins record CRDTs for fields. These can however be substituted with other CRDT types as needed.

So far, we can run all YCSB workloads on Flec, and the results look promising. The remaining work consists of running the benchmarks in different configurations and comparing the results with the performance of other database systems.

## 7.5 Closing Remarks

In this dissertation, we presented a systematic and open approach to the implementation and design of CRDTs. We introduced Flec, a framework that offers a modular approach to developing CRDTs through an open implementation that reifies the replication and convergence process.

We introduced a mechanism to eagerly determine causal stability, enabling early metadata removal in a systematic way. We then adapt this mechanism to support networks where new peers can dynamically join and show how peers can obtain correct states asynchronously.

Next, we introduced an extension to pure operation-based CRDTs that improves responsiveness by allowing pending operations stored in the RCB buffer to be partially applied before all causal dependencies have arrived.

Finally, we proposed Nested Pure Operation-Based CRDTs, a framework for building nested replicated data structures. Our approach allows CRDTs to support composition without semantic changes or structural limitations.

All these techniques and mechanisms were implemented on top of Flec. To conclude, our designs come together to form a reference CRDT framework implementation, where the implementation closely follows the provided specifications. We believe that our approach is a step forward in enabling the use of general-purpose CRDTs in a wide variety of applications.



# Appendix A

## DFS Code Listings

This appendix contains code listings with portions from our distributed filesystem test implementation. A legend for the used types can be found in Table A.1.

Table A.1: Legend for the TypeScript classes and types used in the DFS implementation.

<b>Class / Type</b>	<b>Description</b>
RWWMap	Nested Remove-Wins Map CRDT.
RUWMap	Nested Update-Wins Map CRDT.
ImmutableCRDT	ImmutableCRDT map. Nested CRDT map that works as a C struct.
Register<T>	LLW-Register CRDT, containing a primitive value of type T.
AccessRightF	Alias of the 'Number' type, represents a bit vector with access flags.
AccessRight	Abstraction over AccessRightF, never stores in a CRDT, just used for easy modification of the access right bit vectors.
OpCRDT	Abstract CRDT class in Flec, for creating operation-based CRDTs.
GroupID / UserID / FileID	Aliases for strings that represent UUIDs.

Listing A.1: Access right class representations.

```
1 class AccessRight {
2     constructor(public admin: boolean, public read: boolean,
3         public write: boolean) {}
4
5     static fromEnum(v : AccessRightF) {
6         return new AccessRight((v & 0b100) > 0,
7             (v & 0b010) > 0,
8             (v & 0b001) > 0);
9     }
10
11     toEnum() : AccessRight {
12         return ((this.admin ? 0b100 : 0) | (this.read ? 0b010 : 0)
13             | (this.write ? 0b001 : 0)) as unknown as AccessRight;
14     }
15 }
16
17 enum AccessRightF {
18     UNone = 0b000,
19     UR     = 0b010,
20     UW     = 0b001,
21     URW    = 0b011,
22     ANone = 0b100,
23     AR     = 0b110,
24     AW     = 0b101,
25     ARW    = 0b111,
26 }
```

---

Listing A.2: DFS CRDT Interface.

```
1 interface FSOperation {
2     ChangeOwner(userId: UserID, newOwnerId: UserID, fileId: NodeID
3     ),
4     ChangeGroup(userId: UserID, newGroupId: GroupID, fileId:
5     NodeID),
6     ChangeGroupPermission(userId: UserID, newPermission:
7     AccessRightF, fileId: NodeID),
8     ChangeOwnerPermission(userId: UserID, newPermission:
9     AccessRightF, fileId: NodeID),
10    ChangeOtherPermission(userId: UserID, newPermission:
11    AccessRightF, fileId: NodeID),
12
13    CreateFile(userId: UserID, groupId: GroupID, fileId: NodeID),
14    WriteFile(userId: UserID, fileId: NodeID),
15    DeleteFile(userId: UserID, fileId: NodeID),
16
17    CreateUser(with_admin_rights: boolean, id: string),
18    CreateGroup(),
19
20    AssignUserToGroup(authorId: UserID, groupId: GroupID, userId:
21    UserID),
22
23    update(key: string)
24 }
```

Listing A.3: The general structure of the DFS nested CRDT, highlighting the main nested children that contain the filesystem meta-data.

```
1 export class DistributedFS extends OpCRDT<FSOperation> {
2   handler: FSOperation;
3   ...
4
5   files = new RRWMap(t => new ImmutableCRDT({
6     access_right_owner: new Register<AccessRightF>(),
7     access_right_group: new Register<AccessRightF>(),
8     access_right_other: new Register<AccessRightF>(),
9     file_owner: new Register<UserID>(),
10    file_group: new Register<GroupID>(),
11    file_data: new Register<string>()
12  }));
13
14  groups = new RRWMap(t => new ImmutableCRDT({
15    group_users: new ASet(), // must be RW
16    created: new Register<flag>()
17  }));
18
19  users = new RUWMap(t => new ImmutableCRDT({
20    is_admin: new Register<flag>()
21  }));
22  ...
23
24  onLoaded() {
25    this.addChild("files", this.files);
26    this.addChild("users", this.users);
27    this.addChild("groups", this.groups);
28  }
29  ... continued on the next page ...
```

---

```

30  setHandler() {
31      const me = this;
32      this.handler = {
33
34          ChangeOwner(userId: UserID, newOwnerId: UserID, fileId:
NodeID)    { ... },
35          ChangeGroup(userId: UserID, newGroupId: GroupID, fileId:
NodeID)    { ... },
36          ChangeOwnerPermission(userId: UserID, newPerm: AR, fileId:
NodeID) { ... },
37          ChangeGroupPermission(userId: UserID, newPerm: AR, fileId:
NodeID) { ... },
38          ChangeOtherPermission(userId: UserID, newPerm: AR, fileId:
NodeID) { ... },
39          ...
40          CreateUser(with_admin_rights: boolean, id: string) { /* ...
*/ },
41          CreateGroup() { /* ... */ },
42          AssignUserToGroup(authorId: UserID, groupId: GroupID, userId
: UserID) { ... },
43          CreateFile(userId: UserID, groupId: GroupID, fileId: NodeID)
{ ... see listing below ... },
44          WriteFile(userId: UserID, fileId: NodeID) { ... },
45          ...
46          update(key: string) { }
47      }
48  }
49 }

```

Listing A.4: Structure of the operation handling code for the DFS. Included is the code for the CreateFile callback, which can either be invoked locally or as a result of a replicated operation.

```
1 setHandler() {
2   const me = this;
3
4   this.handler = {
5     ...
6
7     CreateFile(userId: UserID, groupId: GroupID, fileId: NodeID) {
8       const user = me.users.lookup(userId) as any;
9       const group = me.groups.lookup(groupId) as any;
10
11      if (group && user && group.group_users.contains(userId)) {
12        console.log("adding file");
13
14        me.files.update([ { key: fileId, op: "update" },
15          { key: "file_owner", op: "write" } ], userId);
16        me.files.update([ { key: fileId, op: "update" },
17          { key: "file_group", op: "write" } ], groupId);
18
19        const isAdmin = user.is_admin.is(FLAG_TRUE);
20        const access_owner = new AccessRight(isAdmin, true, true);
21        const access_group = new AccessRight(isAdmin, true, false);
22        const access_other = new AccessRight(isAdmin, true, false);
23
24        this.files.update([ { key: fileId, op: "update" },
25          { key: "access_right_owner", op: "write" } ], access_owner.
26          toEnum());
27        this.files.update([ { key: fileId, op: "update" },
28          { key: "access_right_group", op: "write" } ], access_group.
29          toEnum());
30        this.files.update([ { key: fileId, op: "update" },
31          { key: "access_right_other", op: "write" } ], access_other.
32          toEnum());
33      }
34    },
35    ...
36  };
37 }
```

---

Listing A.5: User API for local mutations to DFS CRDT, allowing simple modification of the DFS meta-data.

```
1  CreateUser(with_admin_rights: boolean) {
2    const id = this.getUID();
3    this.performOp("CreateUser", [with_admin_rights, id]);
4    return id;
5  };
6
7  CreateGroup() {
8    const id = this.getUID();
9    this.performNestedOp("update", [{ key: "groups", op: "update"
10   },
11   { key: id, op: "update" },
12   { key: "created", op: "write" }], [FLAG_TRUE]);
13   return id;
14 };
15
16 CreateFile(userId: UserID, groupId: GroupID) {
17   const id = this.getUID();
18   this.performOp("CreateFile", [userId, groupId, id]);
19   return id;
20 }
...

```

Listing A.6: Example test code for the DFS CRDT, which creates a new admin user, a new group, adds the user to a group, and then creates and writes a file with this new user.

```
1  test() {
2    const userId = this.CreateUser(true);
3    const groupId = this.CreateGroup();
4
5    this.performOp("AssignUserToGroup", [userId, groupId, userId])
6    ;
7
8    const fileId = this.CreateFile(userId, groupId);
9    this.performOp("WriteFile", [userId, fileId]);
10 }

```



## Appendix B

# A Remove-Wins Pure Operation-Based CRDT

In this appendix chapter, we explore our implementation of a pure operation-based remove-wins set (RW-Set) CRDT implemented in Flec, following the specification in Section 2.3.2.

The implementations of `isRedundantByLog` and `isRedundantByOperation` are a 1-on-1 mapping with the described semantics for the  $R$  and  $R_{\_}$  relations described in Section 2.3.2. When an entry from the log becomes stable, `setEntryStable` (line 16-19) will remove it from the log and place it in a small compacted set `compactState`. The method `newOperation` makes sure that the local compacted set stays up to date when the log changes. The `toSet` method returns a serialised state for the replica by combining the compacted set with the data from the main log.

## APPENDIX B. A REMOVE-WINS PURE OPERATION-BASED CRDT

---

Listing B.1: RW-Set implementation in Flec.

```
1 export class RWSet extends PureOpCRDT<SetOperations> {
2   compactState: Set<string> = new Set();
3
4   protected isArrivingOperationRedundant(entry : SetEntry) {
5     return entry.isAdd() && !!this.log.find(e => e.isRemove()
6     && e.hasSameArgAs(entry) && e.isConcurrent(entry));
7   }
8   protected isPrecedingOperationRedundant(existing: SetEntry,
9   arriving: SetEntry, isRedundant: boolean) {
10    return existing.hasSameArgAs(arriving);
11  }
12  protected isConcurrentOperationRedundant(existing: SetEntry,
13  arriving: SetEntry, isRedundant: boolean) {
14    return arriving.isRemove() && existing.isAdd() && existing
15    .hasSameArgAs(arriving);
16  }
17
18  setEntryStable(entry : SetEntry) : boolean {
19    if (entry.isAdd()) this.compactState.add(entry.args[0]);
20    return true;
21  }
22
23  newOperation = (entry: SetEntry) => this.compactState.delete(
24  entry.args[0]);
25
26  toSet() {
27    const set = new Set(this.compactState);
28    this.getLog().forEach(entry => {
29      if (entry.isAdd())
30        set.add(entry.args[0])
31    });
32    return set;
33  }
34  add(element){
35    this.perform.add(element);
36  }
37  remove(element) {
38    this.perform.remove(element);
39  }
40 }
```

## Appendix C

# Implementation Details for Eager Stability Determination

In this appendix chapter, we explore our implementation of the operation-based framework with eager stability determination from Chapter 4.

Our pure operation-based framework with eager stability determination is implemented on top of Flec, extending on the definitions specified in Table 3.5. Listing C.1 shows the general structure of the pure operation-based CRDT implementation in Flec, with general code redacted.

The `PureOpCRDT` class keeps track of several state variables, the most important being the log. The log is updated when *non-redundant* operations arrive at a replica. To this end, the `onOperation` method is overridden. Listing C.2 shows how `onOperation` updates the log. The `onOperation` method is called every time the RCB layer has to deliver an operation, which can originate either from a local or remote `performOperation` invocation.

The `onOperation` method relies on results of the abstract `isRedundantByOperation` and `isRedundantByLog` methods to determine what entries are added or removed from the log. These methods define the redundancy relations for pure operation-based CRDTs and must be implemented by the CRDT implementor. An example of this can be seen in Appendix B where we implement an RW-Set using the framework.

## APPENDIX C. IMPLEMENTATION DETAILS FOR EAGER STABILITY DETERMINATION

---

Listing C.1: Structure of the PureOpCRDT class, used to implement pure-operation based CRDTs.

```
1 export abstract class PureOpCRDT<O> extends OpCRDT<O> {
2   // CRDT state
3   log : PLogEntry<O>[] = [];
4   compact = {};
5
6   // CRDT network
7   network = [];
8   joinNode: FarRef<this>;
9
10  // used to set stability trigger level
11  logCompactSize : number = 100;
12
13  constructor () { ... }
14
15  // Handle replica joins
16  onNewReplica(ref: FarRef<this>, refs) { ... }
17  onReqJoin (...){ ... }
18  onReqLink (...){ ... }
19  onReqState(...){ ... }
20  getState (...){ ... }
21  getNetwork(...){ ... }
22  setupState(...){ ... }
23
24
25  // Handle new operations
26  onOperation(clock: VectorClock, op: O, args: any[]) { ... }
27
28  // Manage cleanup of causally stable entries
29  getConcurrentEntries(entry: PLogEntry<O>) { ... }
30  markStable () { ... }
31  compactStable(){ ... }
32  gcStable () { ... }
33  cleanup () { ... }
34  setGCParams(logSize: number, intervalSize) { ... }
35
36
37  // Hooks for implementors of pure-op based CRDTs
38  protected setEntryStable( entry: PLogEntry<O> ) : boolean { ... };
39
40  protected removeEntry( entry: PLogEntry<O>) {};
41  protected newOperation(entry: PLogEntry<O>) {};
42
43  protected isRedundantByOperation      (... ) { ... };
44  protected isPrecedingOperationRedundant (... ) { ... };
45  protected isConcurrentOperationRedundant(... ) { ... };
46  protected isRedundantByBufferedOperation(... ) { ... };
47  protected isArrivingOperationRedundant (... ) { ... };
```

---

Listing C.2: The onOperation method is used to process received operations.

```
1 onOperation(clock: VectorClock, op: O, args: any[]) {
2     let entry = new PologEntry<O>(clock, op, args);
3
4     this.newOperation(entry);
5     let isRedundant = this.isRedundantByLog(entry);
6
7     for (let i=this.log.length-1; i>=0; i--) {
8         let e = this.log[i];
9         if (this.isRedundantByOperation(e, entry, isRedundant)) {
10            this.removeEntry( this.log[i] );
11            delete this.log[i];
12        }
13    }
14
15    this.log = this.log.filter(e => typeof e !== "undefined");
16
17    if (!isRedundant) {
18        this.log.push(entry);
19    }
20
21    this.cleanup();
22 }
```

## APPENDIX C. IMPLEMENTATION DETAILS FOR EAGER STABILITY DETERMINATION

---

Listing C.3: Methods allowing instrumentation of the cleanup process.

```
1 cleanup() {
2     if (this.log.length === this.logCompactSize)
3         this.performPendingStableMsg();
4 }
5
6 setGCParams(logSize: number, intervalSize) {
7     this.logCompactSize = logSize;
8     this.setStableMsgInterval(intervalSize);
9
10    this.cleanup();
11 }
```

Listing C.4 shows how the framework marks and compacts causally stable log entries. The `gcStable` method is invoked by the RCB layer whenever some operations are processed. It ensures that periodically all causally stable log entries are marked as stable (`markStable`) and eventually compacted (`compactStable`). As such, it ensures that an entry will *only* be removed once all concurrent entries are stable as well.

A final aspect of our pure-operation based CRDT implementation is the `cleanup` method, which is invoked at the end of `onOperation` as can be seen in Listing C.2.

The `cleanup` method checks the size of the log, and if it is higher than a certain limit it will ask the RCB layer to send any pending stability messages (employing `performPendingStableMsg()`).

---

Listing C.4: The logic used to mark and compact causally stable log entries.

```
1 markStable(){
2     let stableItems = false;
3
4     this.log.forEach(e => {
5         if (this.isCausallyStable(e.clock)) {
6             e.setStable();
7             stableItems = true;
8         }
9     });
10
11     return stableItems;
12 }
13
14 compactStable(){
15     this.log.filter(e => e.isStable && this.getConcurrentEntries(e
16 )
17         .map(e=>e.entry.stable)
18         .reduce((a,b)=> a && b, true))
19         .forEach(e => {
20             if (this.setEntryStable(e))
21                 delete this.log[this.log.indexOf(e)];
22         });
23     this.log = this.log.filter(e => typeof e !== "undefined");
24 }
25
26 gcStable() {
27     if (this.markStable())
28         this.compactStable();
29 }
```

APPENDIX C. IMPLEMENTATION DETAILS FOR EAGER  
STABILITY DETERMINATION

---

# Bibliography

- [ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: a communication subsystem for high availability. In *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84, 1992.
- [Ant] AntidoteDB. <https://www.antidotedb.eu/>. Accessed: 2024-06-01.
- [ASB14] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based CRDTs by decomposition. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency, PaPEC '14*, pages 3:1–3:2, New York, NY, USA, 2014. ACM.
- [ATB<sup>+</sup>16] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 405–414, 2016.
- [Aut] Automerge. Automerge CRDT library. <https://automerge.org/>. Accessed: 2024-06-01.
- [BAL16] Carlos Baquero, Paulo Sérgio Almeida, and Carl Lerche. The problem with embedded CRDT counters and a solution. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC*

- '16, New York, NY, USA, 2016. Association for Computing Machinery.
- [BAS14] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based CRDTs operation-based. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, New York, NY, USA, 2014. Association for Computing Machinery.
- [BAS17] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Pure operation-based replicated data types. *CoRR*, abs/1710.04469, 2017.
- [BDPGB23] Jim Bauwens, Kevin De Porre, and Elisa González Boix. Towards improved collaborative text editing CRDTs by using natural language processing. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '23, page 51–55, New York, NY, USA, 2023. Association for Computing Machinery.
- [BFG<sup>+</sup>12] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.
- [BFLW12] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 283–307, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [BGB19] Jim Bauwens and Elisa González Boix. Memory efficient CRDTs in dynamic environments. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2019, page 48–57, New York, NY, USA, 2019. Association for Computing Machinery.
- [BGB20a] Jim Bauwens and Elisa González Boix. Flec: A versatile programming framework for eventually consistent systems. In

- Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '20, New York, NY, USA, 2020. Association for Computing Machinery.
- [BGB20b] Jim Bauwens and Elisa González Boix. From causality to stability: Understanding and reducing meta-data in CRDTs. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*, MPLR '20, page 3–14, New York, NY, USA, 2020. Association for Computing Machinery.
- [BGB21] Jim Bauwens and Elisa González Boix. Improving the reactivity of pure operation-based CRDTs. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '21, New York, NY, USA, 2021. Association for Computing Machinery.
- [BGB23] Jim Bauwens and Elisa González Boix. Nested Pure Operation-Based CRDTs. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:26, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [BGYZ14] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. volume 49, page 271–284, New York, NY, USA, jan 2014. Association for Computing Machinery.
- [BHG86] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986.
- [BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, jan 1987.
- [BM99] Carlos Baquero and Francisco Moura. Using structural characteristics for autonomous operation. *SIGOPS Oper. Syst. Rev.*, 33(4):90–96, oct 1999.

- [Bre00] Eric Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.
- [BZP<sup>+</sup>12] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balesgas, and Sérgio Duarte. An optimized conflict-free replicated set. *arXiv preprint arXiv:1210.3368*, 2012.
- [CMGB<sup>+</sup>07] Tom Van Cutsem, Stijn Mostinckx, Elisa González Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *XXVI International Conference of the Chilean Society of Computer Science (SCCC'07)*, pages 3–12, Iquique, Chile, 2007.
- [Cos] CosmosDB. <https://azure.microsoft.com/en-us/products/cosmos-db/>. Accessed: 2024-06-01.
- [CST<sup>+</sup>10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.
- [DHJ<sup>+</sup>07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.
- [DPFGB23] Kevin De Porre, Carla Ferreira, and Elisa González Boix. Verifx: Correct replicated data types for the masses. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, WA*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.
- [DVCM<sup>+</sup>06] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D’Hondt, and Wolfgang De Meuter. Ambient-oriented

- programming in ambienttalk. In Dave Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, pages 230–254, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [EG89] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, page 399–407, New York, NY, USA, 1989. Association for Computing Machinery.
- [EKMS19] Nafise Eskandani, Mirko Köhler, Alessandro Margara, and Guido Salvaneschi. Distributed object-oriented programming with multiple consistency levels in consyst. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2019, page 13–14, New York, NY, USA, 2019. Association for Computing Machinery.
- [Ele] ElectricSQL. <https://electric-sql.com/>. Accessed: 2024-06-01.
- [GL02] Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.
- [Gol92] R.A. Golding. Weak consistency group communication for wide-area systems. In *[1992 Proceedings] Second Workshop on the Management of Replicated Data*, pages 13–16, 1992.
- [GPGP18] José Rolando Guay Paz and José Rolando Guay Paz. Introduction to Azure Cosmos DB. *Microsoft Azure Cosmos DB Revealed: A Multi-Model Database Designed for the Cloud*, pages 1–23, 2018.
- [Gra78] J. N. Gray. *Notes on data base operating systems*, pages 393–481. Springer Berlin Heidelberg, Berlin, Heidelberg, 1978.
- [GWB91] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. Clos: integrating object-oriented and functional programming. *Commun. ACM*, 34(9):29–38, sep 1991.

- [HTSC08] Urs Hunkeler, Hong Linh Truong, and Andy J. Stanford-Clark. MQTT-S - a publish/subscribe protocol for wireless sensor networks. In Sunghyun Choi, Jim Kurose, and Krithi Ramamritham, editors, *COMSWARE*, pages 791–798. IEEE, 2008.
- [HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990.
- [KB17] Martin Kleppmann and Alastair R Beresford. A conflict-free replicated JSON datatype. *IEEE Transactions on Parallel & Distributed Systems*, 28(10):2733–2746, oct 2017.
- [KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.
- [Klo10] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.
- [KPSJ19] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.
- [KWvHM19] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 154–178, New York, NY, USA, 2019. Association for Computing Machinery.
- [Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
- [Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

- [LC18] Iraklis Leontiadis and Reza Curtmola. Secure storage with replication and transparent deduplication. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY '18*, page 13–23, New York, NY, USA, 2018. Association for Computing Machinery.
- [LHCW18] Xiao Lv, Fazhi He, Yuan Cheng, and Yiqi Wu. A novel CRDT-based synchronization method for real-time collaborative cad systems. *Advanced Engineering Informatics*, 38:381 – 391, 2018.
- [LPS10] Mihai Letia, Nuno Preguiça, and Marc Shapiro. Consistency without concurrency control in large, dynamic systems. *SIGOPS Oper. Syst. Rev.*, 44(2):29–34, April 2010.
- [LSB<sup>+</sup>19] Pedro Lopes, João Sousa, Valter Balegas, Carla Ferreira, Sérgio Duarte, Annette Bieniusa, Rodrigo Rodrigues, and Nuno M. Preguiça. Antidote SQL: relaxed when possible, strict when necessary. *CoRR*, abs/1902.03576, 2019.
- [MMF01] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In Yair Frankel, editor, *Financial Cryptography*, pages 349–378, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
- [MVR15] Christopher Meiklejohn and Peter Van Roy. Lasp: A Language for Distributed, Coordination-free Programming. In *17th Int. Symp. on Principles and Practice of Declarative Programming, PPDP '15*, pages 184–195, 2015.
- [Myt19] Myter, Florian. *Triumvirate : a programming language design for distributed rich internet applications*. PhD thesis, Ghent University, 2019.
- [NJDK16] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. pages 39–49, 11 2016.
- [OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual*

- Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.
- [PMSL09] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403, 2009.
- [Pre18] Nuno Preguiça. Conflict-free replicated data types: An overview, 2018.
- [RB94] Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Trans. Program. Lang. Syst.*, 16(3):986–1009, may 1994.
- [RJKL11] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354 – 368, 2011.
- [Sha17] Marc Shapiro. Replicated Data Types. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia Of Database Systems*, volume Replicated Data Types, pages 1–5. Springer-Verlag, July 2017.
- [SPBZ11a] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.
- [SPBZ11b] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400, Grenoble, France, October 2011. Springer.
- [TTP<sup>+</sup>95] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29(5):172–182, 1995.

- [TvS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.
- [Vog08] Werner Vogels. Eventually consistent. *Queue*, 6(6):14–19, October 2008.
- [WB84] Gene T.J. Wu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, page 233–242, New York, NY, USA, 1984. Association for Computing Machinery.
- [WB18] Mathias Weber and Annette Bieniusa. Acgregate: A framework for practical access control for applications using weakly consistent databases, 2018.
- [WMM20] Matthew Weidner, Heather Miller, and Christopher Meiklejohn. Composing and decomposing op-based CRDTs with semidirect products. August 2020.
- [WQK<sup>+</sup>23] Matthew Weidner, Huairui Qi, Maxime Kjaer, Ria Pradeep, Benito Geordie, Yicheng Zhang, Gregory Schare, Xuan Tang, Sicheng Xing, and Heather Miller. Collabs: A flexible and performant CRDT collaboration framework, 2023.
- [WUM10] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Trans. on Parallel and Distributed Systems*, 21(8):1162–1174, August 2010.
- [YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in abcl/1. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOP-SLA '86, page 258–268, New York, NY, USA, 1986. Association for Computing Machinery.
- [Yjs] Yjs shared editing. <https://yjs.dev/>. Accessed: 2024-06-01.

## BIBLIOGRAPHY

---

- [You22] Georges Younes. *Dynamic End-to-End Reliable Causal Delivery Middleware For Geo-Replicated Services*. PhD thesis, 2022.
- [YYRB21] Elena Yanakieva, Michael Youssef, Ahmad Hussein Rezae, and Annette Bieniusa. Access control conflict resolution in distributed file systems using CRDTs. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC '21*, New York, NY, USA, 2021. Association for Computing Machinery.
- [ZBPH14] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of CRDTs. In E. Abraham and C. Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 33–48, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.