SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Design and Verification of Byzantine Fault Tolerant CRDTs

Liangrun Da

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Design and Verification of Byzantine Fault Tolerant CRDTs

Author:	Liangrun Da
Examiner:	Bhatotia Pramod
Supervisor:	Martin Kleppmann
Submission Date:	17.10.2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 17.10.2024 Liangrun Da

Acknowledgments

I want to thank my supervisor, Martin Kleppmann, for his patient guidance and valuable feedback during the whole process of my master thesis project. His insightful comments and suggestions have greatly improved the quality of this thesis. I also appreciate the support from Dominic Mulligan, Lukas Stevens and Tobias Nipkow for their suggestions on the Isabelle formalization. Also, I would like to thank Bhatotia Pramod for his help on the thesis project formalities.

I am grateful to my parents for their continuous support and encouragement throughout my studies. I would also like to thank my girlfriend for her companionship and constant encouragement, which have been invaluable throughout this journey.

Abstract

Conflict-Free Replicated Data Types (CRDTs) are designed to ensure consistency of replicated data in peer-to-peer settings. However, their consistency guarantees are compromised in the presence of untrusted nodes that may deviate from the protocol, a common scenario in real-world peer-to-peer systems. This thesis addresses the issue of maintaining CRDT consistency in Byzantine environments. We propose a generic framework that can be used to retrofit Byzantine Fault Tolerance (BFT) to existing CRDTs with minimal modifications. Our framework employs mechanized proofs using the Isabelle/HOL proof assistant with minimal assumptions, only requiring the existence of a collision-resistant cryptographic hash function. The framework provides a verification basis that allows for verification of the resulting BFT CRDTs under a wide range of potential attack scenarios. We demonstrate the practicality of our approach by successfully designing and verifying Byzantine Fault Tolerant versions of Observed-Remove Set (ORSet) and Replicated Growable Array (RGA), two widely-used CRDTs.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**

2 Background **4**
2.1 Conflict-free Replicated Data Types (CRDTs) 4
2.2 An introduction to Isabelle/HOL 4
2.2.1 Types and Terms 5
2.2.2 Datatypes 5
2.2.3 Functions 6
2.2.4 Theorems and Proofs 6
2.2.5 Inductive Predicates 6
2.2.6 Locale 7

3 Related Work **8**
3.1 BFT consensus 8
3.2 A framework for verifying CRDTs in Isabelle/HOL 8
3.2.1 Abstract Convergence 9
3.2.2 Observed-Remove Set (ORSet) 11
3.2.3 Replicated Growable Array (RGA) 13

4 System Model **16**
4.1 Participants and Network Model 16
4.2 Attacker Model 16
4.3 Correctness Model 17

5 Known Attacks on CRDTs **19**
5.1 Eventual Delivery Attack 19
5.2 CRDT-specific Attacks 20
5.2.1 ORSet 21
5.2.2 RGA 22

6 System Design and Formalization **25**
6.1 System Overview . . . 25
6.2 Hash DAG . . . 26
6.2.1 Definition of Hash Graph . . . 26
6.2.2 Heads, Predecessors and Ancestors . . . 26
6.2.3 Structural Validity . . . 27
6.2.4 Properties of Head Nodes . . . 29
6.2.5 Properties of Reachable Nodes . . . 30
6.3 BFT Convergence . . . 34
6.3.1 Behavior of Correct Peers . . . 36
6.3.2 Properties of a Correct Peer . . . 37
6.3.3 Convergence . . . 39
6.3.4 Synchronization . . . 39
6.3.5 Definition of valid sequence . . . 41
6.3.6 Synchronization Proof . . . 42
6.3.7 BFT Strong Eventual Consistency . . . 44
6.4 Common Vulnerabilities . . . 45
6.4.1 Uniqueness of IDs . . . 45
6.4.2 Dependency between Operations . . . 46
6.5 Conclusion . . . 46

7 Evaluation **48**
7.1 Methodology . . . 48
7.2 Byzantine Fault Tolerant ORSet . . . 49
7.2.1 Concurrent Operations Commute . . . 50
7.2.2 Synchronization . . . 53
7.2.3 No Failure . . . 53
7.2.4 BFT Strong Eventual Consistency . . . 53
7.3 Byzantine Fault Tolerant RGA . . . 54
7.3.1 Concurrent Operations Commute . . . 56
7.3.2 Synchronization . . . 60
7.3.3 No Failure . . . 60
7.3.4 BFT Strong Eventual Consistency . . . 61
7.4 Conclusion . . . 61

8 Conclusion **63**
8.1 Summary . . . 63
8.2 Future Work . . . 64

List of Figures **65**

Bibliography **66**

1 Introduction

Conflict-free Replicated Data Types (CRDTs) have long been considered a promising solution for achieving *strong eventual consistency (SEC)* in peer-to-peer systems [Sha+11b]. While *Operational Transformation (OT)* can also achieve SEC, it typically requires a central server to resolve conflicts [Nic+95; Day10; App11; Wan+15]. In contrast, CRDTs can operate in a fully decentralized manner, making them particularly well-suited for peer-to-peer systems. However, this advantage comes with a critical assumption that all the participants in the system are honest and strictly follow the protocol. Unfortunately, this assumption often fails to hold in real-world peer-to-peer environments.

A key characteristic of most P2P systems is the absence of centralized control, allowing anyone to join or leave the system freely, making it impossible to guarantee that all participants will adhere to the protocol. A peer might deviate from the protocol either unintentionally (due to hardware failures or software bugs) or intentionally (with malicious intent). Regardless of the cause, any peer that fails to follow the protocol is called a *Byzantine-faulty* peer [LSP19]. Malicious peers may send false, malformed, or contradictory messages to other peers, either to gain unfair advantages or simply to disrupt the system's guarantees. A malicious peer might pretend to be a legitimate peer, and it is difficult to detect malicious behaviors in a P2P system. Even if a peer is detected as malicious and excluded from the system, the peer might change its identity and rejoin the system. Moreover, once Byzantine activity has occurred, non-BFT algorithms lack the mechanisms to roll back or undo the effects of malicious updates, potentially leaving the system in a permanently compromised state.

This fundamental conflict between the honesty assumptions of CRDTs and the reality of P2P systems has been a major obstacle to the adoption of CRDTs in P2P systems. Most existing CRDT algorithms are not suited for open P2P systems that anyone can join, limiting their use to small and trusted groups. Previous attempts to apply CRDTs to "massive-scale" editing systems [And+13; Lv+16; WUM09] and public editing systems [WUM09; NMM16; WUM07] have overlooked the challenges posed by Byzantine peers, as the openness and scale of the systems make them particularly vulnerable to attacks.

Even in controlled environments with a small number of trusted users, a peer might still be compromised, e.g., by malware. This can lead to serious consequences. For example, when a group of users are collaborating on a shared contract document, a malicious attacker could manipulate the system to show different contract contents to

different users. The compromised peer could silently carry out such attacks without being detected unless the users explicitly compare their versions of the document, which is a significant overhead if users need to verify the contents of all their shared documents periodically. Even worse, once detected, the malicious updates might be difficult to remove from the document. For example, a non-malicious edit might be positioned relative to text introduced by a malicious update. Removing the malicious update would cause the non-malicious edit to be misplaced or become nonsensical. This intricate web of dependencies makes it challenging to cleanly remove malicious content without potentially corrupting or losing valid contributions to the document.

To make use of CRDTs in open peer-to-peer systems and to mitigate the impact of device compromises, it is essential to implement Byzantine Fault-Tolerant (BFT) CRDTs. These BFT CRDTs maintain the strong eventual consistency guarantees even in the presence of Byzantine-faulty peers. This enables CRDTs to function reliably in open, decentralized environments where not all participants can be trusted to follow the protocol correctly.

However, implementing BFT CRDTs presents several challenges:

- Given the wide variety of existing CRDT algorithms — each with distinct performance characteristics and semantics, even within the same data structure type, such as the numerous CRDTs designed for text editing [Ost+06; Roh+11; Pre+09; WUM09; Néd+13; Lit+22; WGK23] — it would be impractical and prohibitively time-consuming to redesign each algorithm from scratch for Byzantine fault tolerance. This diversity necessitates a generic approach that can be applied across different CRDT types, rather than creating individual BFT versions for each CRDT algorithm.

- It is challenging to reason about the correctness of an algorithm in a distributed system, even a non-Byzantine one, as the network is unpredictable. Messages may be delayed, duplicated, lost, or delivered out of order. As a result, it becomes difficult to reason about the correctness of algorithms in distributed systems. Many algorithms may seem plausible at first but were later found to be flawed. This is not an overstatement; it has indeed occurred in peer-reviewed publications from time to time. For example, in the last decades, many OT algorithms were found to be flawed. The original publications of dOPT [EG89], adOPTed [RNG96], IMOR [Imi+03], SOCT2 [SCF98], and SDT [LL04] all claim that their algorithms could achieve SEC. However, subsequent publications provided counterexamples that proved these claims to be incorrect [Imi+03; Imi+06; Ost+05]. Some of these algorithms had hand-written proofs, but these proofs were later shown to be wrong. Other algorithms even had machine-checked proofs, but were later shown to have used incorrect assumptions.

- It is even more complicated to reason about the correctness of an algorithm in a Byzantine environment, where the adversary behavior is unpredictable. A malicious peer might combine multiple existing tactics to attack the system, or use a novel approach that was not considered in the previous studies. Therefore, we need a model that can simulate arbitrary attacks to reason about the correctness of the algorithm.

Addressing these challenges would significantly expand the applicability of CRDTs, enabling their use in a wider range of decentralized systems and enhancing their resilience against both unintentional failures and malicious attacks.

Our contributions are as follows:

- We propose a generic framework that can be used to retrofit BFT to existing CRDTs with minimal modifications to the original CRDT algorithms. The framework is based on the idea of using a Hash Directed Acyclic Graph (HashDAG) to ensure that honest nodes deliver the same set of updates in causal order, and provide a foundation for making consistent desicions on the validity of incoming updates from other peers.
- The framework is fully formalized and verified using Isabelle/HOL, a machine-checking proof assistant. We only assume the existence of a collision-resistant cryptographic hash function, which is a widely accepted and used assumption in the cryptographic research community. Our work avoids the potential flaws that have been found in hand-written proofs or machine-checked proofs with incorrect assumptions in the past. Furthermore, the framework provides a verification basis that allows for verification of the resulting BFT CRDTs under a wide range of potential attack scenarios.
- To evaluate our framework, we retrofit BFT to two common CRDTs, Observed-Remove Set (ORSet) [Sha+11a] and Replicated Growable Array (RGA) [Roh+11], and verify the correctness of the resulting BFT CRDTs. ORSet is a CRDT that allows adding and removing elements from a set, supported as a primitive by the Lasp language [MV15]. RGA is a sequence CRDT that allows inserting and deleting elements in a list. Popular CRDT libraries such as Automerge [Aut] use RGA as the underlying CRDT for text editing.

Our Isabelle files are open-sourced on GitHub [DK24]. This allows others to quickly verify the correctness of our work with a single command, as well as build their own proofs on top of our framework.

2 Background

2.1 Conflict-free Replicated Data Types (CRDTs)

Conflict-free Replicated Data Types are a class of data types that can be replicated across multiple nodes in a distributed system without the need for central coordination between the nodes. CRDTs are designed to ensure that the replicas converge to the same state if they receive the same set of updates, regardless of the order in which the updates are applied. This property is known as Strong Eventual Consistency (SEC) [Sha+11b].

CRDTs can be classified into three categories: operation-based CRDTs, state-based CRDTs and delta-based CRDTs. Operation-based CRDTs propagate the operations to other replicas, while state-based CRDTs propagate the entire state of the data structure. Delta-based CRDTs propagate the deltas, which are the differences between the states. Op-based CRDTs are more efficient in terms of network bandwidth but require more space to store the operation histories.

Numerous CRDTs have been proposed, and these data types resemble traditional non-collaborative data structures, such as counters, registers [Sha+11a], maps [BAL16], sets [Bie+12], XML [MUW10], JSON [KB17], etc. These basic CRDTs can be used as building blocks to construct collaborative applications. Furthermore, they can be combined to create more complex CRDTs [Kue+23].

For collaborative text editing, various CRDT algorithms have been introduced, such as WOOT [Ost+06], RGA [Roh+11], and Eg-walker [GK24]. These algorithms vary in terms of performance and semantics, enabling applications to select a CRDT that aligns with their specific desired behavior. CRDT libraries offer implementations of these algorithms. For instance, Automerge [Aut] is a JSON CRDT library and the text type embedded in the JSON document is implemented using the RGA algorithm. Yjs is an implementation of the YATA algorithm [Nic+16].

2.2 An introduction to Isabelle/HOL

Isabelle is a generic proof assistant that supports various logics, including higher-order logic (HOL), Zermelo-Fraenkel set theory (ZF) and others. Informally, Isabelle/HOL

is the combination of logic and functional programming [Nip13], making it a powerful and expressive tool for formalizing computer science algorithms. For instance, Isabelle/HOL was used to verify the correctness of the L4 microkernel [Kle+14]. Although Isabelle/HOL doesn't come with any built-in support for distributed systems, the logic is expressive enough to model the behavior of distributed systems. Previously, Gomes et al. [Gom+17b] built a framework on top of Isabelle/HOL to verify the strong eventual consistency of CRDTs.

In this section, we provide a brief introduction to the key concepts of Isabelle/HOL along with some examples. For readers interested in learning more about Isabelle/HOL, we recommend referring to "Programming and Proving in Isabelle/HOL" by Tobias Nipkow [Nip13].

2.2.1 Types and Terms

In Isabelle/HOL, types are used to classify terms. For instance, the type *nat* represents the natural numbers, and the type *bool* represents the boolean values. Terms are the basic building blocks of Isabelle/HOL. For instance, the term *0* represents the natural number zero, and the term *True* represents the boolean value true.

2.2.2 Datatypes

Following the ML-style syntax, new datatypes are defined using the *datatype* keyword. The general syntax for defining a datatype is as follows:

$$\begin{array}{lcl} \textbf{datatype}\ ('a_1,\ldots,'a_n)t & = & C_1\ \tau_{1,1}\ldots\tau_{1,n_1} \\ & | & \ldots \\ & | & C_k\ \tau_{k,1}\ldots\tau_{k,n_k} \end{array}$$

Each constructor of a datatype is defined as a function that takes the appropriate number of arguments and returns the constructed type. The general syntax for defining a constructor is as follows:

$$C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow ('a_1,\ldots,'a_n)t$$

For instance, the datatype *tree* can be defined as follows:

datatype *'a tree = Leaf | Node ‹'a tree› ‹'a› ‹'a tree›*

This definition introduces a new type *'a tree* that represents a binary tree with nodes of type *'a*. The datatype *tree* has two constructors: *Leaf* and *Node*. The *Leaf* constructor represents an empty tree, while the *Node* constructor represents a non-empty tree with a left subtree, a value, and a right subtree.

2.2.3 Functions

In Isabelle/HOL, functions are defined using the *fun* keyword. For instance, the function *size* that computes the size of a binary tree can be defined as follows:

fun *size* :: ‹*'a tree* ⇒ *nat*› **where**
‹*size Leaf* = *0*›
‹*size* (*Node l v r*) = *1* + *size l* + *size r*›

This definition introduces a new function *size* that takes a binary tree as input and returns the size of the tree. The function *size* is defined recursively: the size of an empty tree is zero, and the size of a non-empty tree is one plus the size of the left subtree plus the size of the right subtree.

2.2.4 Theorems and Proofs

lemma, *theorem* and *corollary* are used interchangeably in Isabelle/HOL to refer to a statement that is proved to be true. For example, the lemma

lemma *size-Leaf*: ‹*size Leaf* = *0*›

states that the size of an empty tree is zero. To prove this lemma, we can use the *by* keyword followed by a proof method. For instance, the proof of the above lemma can be written as follows:

lemma *size-Leaf*: ‹*size Leaf* = *0*›
by *simp*

This proof uses the *simp* method, which simplifies the goal using the simplification rules of Isabelle/HOL.

2.2.5 Inductive Predicates

In Isabelle/HOL, inductive predicates are used to define properties in a inductive way. For instance, the property of being a binary search tree can be defined as an inductive predicate as follows:

inductive *is-bst* :: ‹*int tree* ⇒ *bool*› **where**
‹*is-bst Leaf*›
‹⟦*is-bst l*; *is-bst r*;
∀ *x* ∈ *set-tree l*. *x* < *a*;
∀ *x* ∈ *set-tree r*. *x* > *a*⟧
⟹ *is-bst* (*Node l a r*)›

The base case states that an empty tree is a binary search tree. The inductive case states that a non-empty binary tree is a binary search tree if its left and right subtrees are binary search trees and the value of the root node is greater than all values in the left subtree and less than the all values in the right subtree.

2.2.6 Locale

A locale is a mechanism in Isabelle/HOL for defining a context in which certain assumptions hold. A simple locale declares a set of parameters using keyword *fix* and a set of assumptions using keyword *assumes* that are relevant to the context. An example of a locale is *partial-order*, which models a partial order by assuming the existence of a reflexive, antisymmetric, and transitive binary relation.

locale *partial-order* =
  fixes *le* :: ‹'*a* ⇒ '*a* ⇒ *bool*› (**infixl** ‹⊑› *50*)
  assumes *refl* [*intro*, *simp*]: ‹*x* ⊑ *x*›
    and *anti-sym* [*intro*]: ‹⟦ *x* ⊑ *y*; *y* ⊑ *x* ⟧ ⟹ *x* = *y*›
    and *trans* [*trans*]: ‹⟦ *x* ⊑ *y*; *y* ⊑ *z* ⟧ ⟹ *x* ⊑ *z*›

The parameter of this locale is *le*, which is a binary predicate with infix syntax ⊑. The assumptions define the properties of the *le* relation: reflexive, antisymmetric, and transitive. In the subsequent context, those assumptions are available as facts that can be used in the proofs.

3 Related Work

3.1 BFT consensus

Byzantine fault-tolerant (BFT) consensus algorithms have been extensively studied due to their use in blockchains. While there are certain similarities between BFT consensus and BFT CRDTs, such as their shared goal of ensuring replica consistency, the problems of BFT CRDTs and BFT consensus are fundamentally different.

Firstly, Byzantine consensus provides significantly stronger guarantees than CRDTs require. BFT consensus guarantees the total order delivery of messages, but CRDTs allow updates to be applied in any causal order. The additional overhead imposed by enforcing total order broadcast is unnecessary for CRDTs.

Secondly, BFT consensus algorithms, including PBFT [CL+99], provably require the faulty peers to be less than $1/3$ of the total peers [DLS88], which makes them susceptible to Sybil attacks [Dou02], and unsuitable for systems that anybody can join. Although Proof-of-work [Nak08] can be used as a countermeasure, it is extraordinarily expensive; and Proof-of-stake [Ban+19] is limited to the field of cryptocurrency because validators are chosen based on the number of staked coins they have, and it is not appropriate for non-financial applications. BFT CRDTs, on the other hand, do not have a limit on the number of Byzantine peers they can tolerate, and therefore no sybil countermeasures are required. The only requirement is that the correct peers are directly or indirectly connected [KH20].

In summary, the problems of BFT consensus and BFT CRDTs are fundamentally different. The stronger guarantees, participant assumptions, system models, and performance characteristics of BFT consensus algorithms make them unsuitable for direct application to BFT CRDT scenarios. Therefore, specialized approaches are needed to address the unique challenges of Byzantine fault tolerance in CRDTs.

3.2 A framework for verifying CRDTs in Isabelle/HOL

Gomes et al. [Gom+17b] developed a framework for verifying the Strong Eventual Consistency (SEC) of CRDTs using Isabelle/HOL. Although this framework assumes all peers are honest and strictly adhere to the protocol, which is not applicable in a

Byzantine environment, some of its theorems can still be reused in our work.

The key concepts in this framework are as follows:

1. Abstract Convergence Theory: The framework begins by proving an abstract convergence theory. This theory makes minimal assumptions, only requiring the existence of a happens-before relation and an abstract interpretation function. It does not make any specific assumptions about the CRDT algorithm or the network model. This abstraction allows the theory to be widely applicable. In our framework, we reuse the abstract convergence theory and extend it to the Byzantine environment.

2. Causal Broadcast Network: The framework then introduces a causal broadcast network. In this context, the happens-before relation is interpreted as the network message happens-before relation as defined by Lamport [Lam78]. Further, the framework provides an extended version of the causal broadcast network with constraint defined by specific CRDTs, where each peer checks the validity of a update before it is broadcast, and this implies all peers have to be honest. The causal broadcast network bridges the gap between the abstract theory and more concrete implementations. We do not reuse network theory in our work, because we focus on the Byzantine environment, where peers might not follow the protocol strictly, i.e. a peer might send an update that is not valid.

3. CRDT Algorithm Implementation: To demonstrate the practical applicability of the framework, it implements and verifies several CRDT algorithms, including GCounter (Grow-only Counter), ORSet (Observed-Remove Set), and RGA (Replicated Growable Array). These implementations serve as concrete examples of how the framework can be used to verify that different peers will converge to the same state, provided they follow the protocol and receive the same set of updates. In our work, we have modified the ORSet and RGA algorithms to make them compatible with our framework and capable of achieving consistency in a Byzantine environment.

In this section, we summarize some of the key definitions and theorems from Gomes et al.'s framework, as our work builds upon these foundations. By introducing these concepts, we establish a common ground for understanding the extensions and modifications we've made to adapt the framework for Byzantine environments.

3.2.1 Abstract Convergence

Abstract Convergence is a locale named *happens-before* that assumes the existance of *hb*, *hb-weak* and *interp*.

The *hb* relation has to be a partial order, while the *hb-weak* relation must be a preorder. *preorder* is a built-in class in Isabelle/HOL. The *interp* function is a state transformer, which takes a state and operation as input and produces a new state or failure. Operations are modeled as a type variable *'a*, and states as *'b*. $\rightharpoonup$ means partial function, and a *None* value indicates failure. It is flexible and can be defined according to the specific CRDT being modeled.

locale *happens-before* = *preorder hb-weak hb*
  for *hb-weak* :: ‹*'a* ⇒ *'a* ⇒ *bool*› (**infix** ‹$\preceq$› *50*)
  and *hb* :: ‹*'a* ⇒ *'a* ⇒ *bool*› (**infix** ‹$\prec$› *50*) +
  fixes *interp* :: ‹*'a* ⇒ *'b* ⇀ *'b*› (‹⟨-⟩› [*0*] *1000*)

Two operations are said to be concurrent if neither of them happens before the other:

definition *concurrent* :: ‹*'a* ⇒ *'a* ⇒ *bool*› (**infix** ‹$\parallel$› *50*) **where**
  ‹*s1* $\parallel$ *s2* ≡ ¬ (*s1* $\prec$ *s2*) ∧ ¬ (*s2* $\prec$ *s1*)›

Inductive predicate *hb-consistent* is used to describe that the operations in a list are ordered according to the *hb* relation, i.e the operations are in causal order. An empty list is *hb-consistent* and any list that is *hb-consistent* extended by an operation that does not happen before any of the existing operations in that list is also *hb-consistent*.

inductive *hb-consistent* :: ‹*'a list* ⇒ *bool*› **where**
  [*intro*!]: ‹*hb-consistent* []›
  [*intro*!]: ‹⟦ *hb-consistent xs*; ∀ *x* ∈ *set xs*. ¬ *y* $\prec$ *x* ⟧ ⟹ *hb-consistent* (*xs* @ [*y*])›

The *kleisli* function is used to compose two state transformers. $\langle x \rangle \rhd \langle y \rangle$ means that we first apply x, and then apply y. If either x or y fails, the entire state transformer fails. $\gg\!=$ is the bind operator for the option monad: if the left-hand side is Some(value), it applies the function on the right-hand side to that value. If the left-hand side is None, it simply returns None without applying the function.

definition *kleisli* :: ‹(*'b* ⇒ *'b option*) ⇒ (*'b* ⇒ *'b option*) ⇒ (*'b* ⇒ *'b option*)› (**infixr** ‹▷› *65*) **where**
  ‹*f* ▷ *g* ≡ λ*x*. (*f x* $\gg\!=$ (λ*y*. *g y*))›

The *apply-operations* function is a function that sequentially combines a list of state transformers. *map interp* is a function that turns a list of operations into a list of state transformers, while *foldl* folds the list of state transformers into a single state transformer.

definition *apply-operations* :: ‹*'a list* ⇒ *'b* ⇀ *'b*› **where**
  ‹*apply-operations es* ≡ *foldl* (▷) *Some* (*map interp es*)›

The *concurrent-ops-commute* predicate holds if for any two concurrent operations x and y, applying x first and then y yields the same result as applying y first and then x. This property implies that the order of applying concurrent operations does not affect the final state.

definition *concurrent-ops-commute* :: ‹*'a list* ⇒ *bool*› **where**
‹*concurrent-ops-commute xs* ≡
∀ *x y*. {*x*, *y*} ⊆ *set xs* ⟶ *concurrent x y* ⟶ ⟨*x*⟩▷⟨*y*⟩ = ⟨*y*⟩▷⟨*x*⟩›

The final theorem of Abstract Convergence states that for two identical sets of updates, if they both satisfy the *concurrent-ops-commute* property and the order of applying these updates is *hb-consistent*, then they will reach the same state. This theorem is straightforward and would likely be accepted without formal proof. While we do not provide a detailed proof here, readers interested in the formal verification process can refer to the source code of this framework available in the Archive of Formal Proofs [Gom+17a].

theorem *convergence*:
assumes ‹*set xs* = *set ys*›
‹*concurrent-ops-commute xs*›
‹*concurrent-ops-commute ys*›
‹*distinct xs*›
‹*distinct ys*›
‹*hb-consistent xs*›
‹*hb-consistent ys*›
shows ‹*apply-operations xs* = *apply-operations ys*›

3.2.2 Observed-Remove Set (ORSet)

The Observed-Remove Set (ORSet) is a CRDT that allows for concurrent additions and removals of elements in a set.

The state of an ORSet is represented as a function. Given an element that might be in the set, this function returns the set of identifiers associated with the operations that added that element to the set and have not been subsequently removed. If *state e* returns an empty set, it means *e* is not present in the set. Otherwise, the set contains the identifiers of all add operations for *e* that have not been overridden by subsequent remove operations, implying that *e* is currently in the set. The state of an ORSet is defined as follows:

type-synonym (*'id*, *'a*) *state* = ‹*'a* ⇒ *'id set*›

The operations in ORSet are defined as follows:

1. *Add i e*: Adds element *e* to the set with a unique identifier *i*. This allows the same element to be added multiple times, as each addition is associated with a distinct identifier.

2. *Rem is e*: Removes element *e* from the set, where *is* is the set of identifiers associated with *e* to be removed.

This definition allows ORSet to handle concurrent additions and removals in a reasonable semantic manner. For instance, a removal operation only removes the specific instances of an element identified by the given set of identifiers, allowing a concurrent add operation to be applied successfully.

The interpretation of operations is to apply the operation immediately to the state. *state ((op-elem oper) := after)* means updating the function "state" so that the element being added or removed is now mapped to the set of IDs "after".

```
definition op-elem :: ‹('id, 'a) operation ⇒ 'a› where
 ‹op-elem oper ≡ case oper of Add i e ⇒ e  Rem is e ⇒ e›
definition interpret-op :: ‹('id, 'a) operation ⇒ ('id, 'a) state ⇀ ('id, 'a) state› (‹⟨-⟩› [0] 1000) where
 ‹interpret-op oper state ≡
   let before = state (op-elem oper);
      after  = case oper of Add i e ⇒ before ∪ {i}  Rem is e ⇒ before − is
   in  Some (state ((op-elem oper) := after))›
```

The *valid-behaviours* function makes sure that the operations are valid before they are broadcast. It enforces that:

1. no two distinct *Add* operations can ever have the same identifer, i.e. the ID is globally unique. In non-Byzantine environments, this can be achieved by using a pair of local incrementing counter and a unique peer ID.

2. A *Rem* operation can only remove identifiers that actually exist in the set. This prevents the removal of non-existent elements.

In the following definition, *i* is the message ID, that is assumed to be unique globally by the causal broadcast protocol. $i = j$ ensures the uniqueness of the IDs for the *Add* operations.

```
definition valid-behaviours :: ‹('id, 'a) state ⇒ 'id × ('id, 'a) operation ⇒ bool› where
 ‹valid-behaviours state msg ≡
   case msg of
     (i, Add j  e) ⇒ i = j
     (i, Rem is e) ⇒ is = state e›
```

In a system with causal broadcast, an operation is delivered after all causally preceding operations have been delivered. If all peers adhere to the *valid-behaviours* rule, any peer delivering a *Rem* operation must have delivered all the *Add* operations it's attempting to remove. Consequently, the scenario of removing a non-existent identifier becomes impossible if all nodes correctly follow the protocol. However, in a Byzantine system this assumption is not safe.

3.2.3 Replicated Growable Array (RGA)

The Replicated Growable Array (RGA) is an ordered list CRDT that supports concurrent insertions and deletions. Each element in the RGA is represented by a tuple containing the element's unique identifier, its value, and a flag indicating whether it has been deleted. An element is defined as follows:

type-synonym $('id, 'v)\ elt = ‹'id \times 'v \times bool›$

Here, *'id* represents the unique identifier type, *'v* is the value type, and the boolean flag indicates whether the element has been deleted (true) or is still present (false).

- **Insert**: Adds a new element to the list after the existing element with ID *id*. An insertion at the head of the list is expressed by using None as the existing element ID. When multiple elements are concurrently inserted in the same place, they are placed in a descending order by their insertion IDs.
- **Delete**: Marks an existing element as deleted by setting its deletion flag to true. The element remains in the structure but is considered removed from the logical view of the list.

The implementation of these operations is shown below. The *insert* function finds the element with the ID *i* and inserts the new element after it. The *insert-body* function is used to skip over the element that have greater IDs than the new element and insert the new element after it. The *delete* function finds the element with the ID *i* and marks it as deleted.

```
fun insert-body :: ‹('id::{linorder}, 'v) elt list ⇒ ('id, 'v) elt ⇒ ('id, 'v) elt list› where
  ‹insert-body []     e = [e]›
  ‹insert-body (x#xs) e =
     (if fst x < fst e then
        e#x#xs
      else x#insert-body xs e)›
fun insert :: ‹('id::{linorder}, 'v) elt list ⇒ ('id, 'v) elt ⇒ 'id option ⇒ ('id, 'v) elt list option› where
  ‹insert xs     e None     = Some (insert-body xs e)›
```

```
  ‹insert []     e (Some i) = None›
  ‹insert (x#xs) e (Some i) =
     (if fst x = i then
        Some (x#insert-body xs e)
      else
        insert xs e (Some i) ⤜ (λt. Some (x#t)))›
fun delete :: ‹('id::{linorder}, 'v) elt list ⇒ 'id ⇒ ('id, 'v) elt list option› where
  ‹delete []              i = None›
  ‹delete ((i', v, flag)#xs) i =
     (if i' = i then
        Some ((i', v, True)#xs)
      else
        delete xs i ⤜ (λt. Some ((i',v,flag)#t)))›
```

The RGA CRDT defines two types of operations that can be transmitted over the network: Insert and Delete. These operations are interpreted by the receiving nodes to update their local state. The interpretation of these operations corresponds to calling the insert and delete functions we discussed earlier. These operations are defined and interpreted as follows:

```
datatype ('id, 'v) operation =
  Insert ‹('id, 'v) elt› ‹'id option›
  Delete ‹'id›
fun interpret-opers :: ‹('id::linorder, 'v) operation ⇒ ('id, 'v) elt list ⇀ ('id, 'v) elt list› (‹⟨-⟩› [0] 1000)
where
  ‹interpret-opers (Insert e n) xs  = insert xs e n›
  ‹interpret-opers (Delete n)   xs  = delete xs n›
```

Similar to the ORSet, the RGA CRDT also defines validity rules to make sure the operations are valid before they are broadcast. The validity rules can be summarized as follows:

1. For an Insert operation:
   - The new element's identifier must be globally unique.
   - If the operation specifies an existing element after which to insert (i.e., it's not inserting at the head), that element's identifier must already exist in the RGA at the time the insert operation is generated.
2. For a Delete operation:
   - The identifier of the element to be deleted must already exist in the RGA.

The implementation of these validity checks is shown in the *valid-rga-msg* function below:

```
definition element-ids :: ‹('id, 'v) elt list ⇒ 'id set› where
‹element-ids list ≡ set (map fst list)›
definition valid-rga-msg :: ‹('id, 'v) elt list ⇒ 'id × ('id::linorder, 'v) operation ⇒ bool› where
‹valid-rga-msg list msg ≡ case msg of
  (i, Insert e  None     ) ⇒ fst e = i
  (i, Insert e (Some pos)) ⇒ fst e = i ∧ pos ∈ element-ids list
  (i, Delete        pos  ) ⇒ pos ∈ element-ids list›
```

Due to the causal order broadcast, when an Insert or Delete operation is delivered, the operation containing the ID it references is guaranteed to have been delivered already if all the peers follow the protocol correctly. Again, this might not hold in a Byzantine environment, as an attacker might send invalid operations, as we will discuss later.

4 System Model

4.1 Participants and Network Model

We assume a peer-to-peer system consisting of arbitrary number of peers, where each peer can follow the protocol or deviate from it. A peer is said to be *Byzantine* if the peer deviates from the protocol, i.e it may do anything, including crashing or malicious behavior. A peer is said to be *correct* if it strictly follows the protocol, but it might be offline or crash. A correct node does not know whether another node is Byzantine or correct. The number of peers in the system might change over time, i.e peers can join or leave the system freely without any restriction.

The network is assumed to be asynchronous and unreliable, where messages can be delayed by an unbounded amount of time, lost, reordered or duplicated. A message can be lost due to network failure, and the receiver cannot distinguish whether a message is lost or delayed. The sender can retry multiple times to send a message until it succeeds. While we do not require that any two correct peers p and q have a direct connection, we assume that all correct peers are at least indirectly connected through other correct peers. In other words, this paper excludes eclipse attacks, where a subset of correct peers is completely isolated from the rest of the network by Byzantine peers, in which case the Byzantine peers can easily prevent the isolated correct peers from receiving any updates from outside the isolated group and vice versa. Moreover, we do not consider denial-of-service or resource exhaustion attacks.

4.2 Attacker Model

Our system considers powerful attackers capable of both active and passive attacks within the network. These attackers can control an arbitrary number of Byzantine peers. However, attackers have no control over the correct peers and they are computationally bounded, so we assume they cannot break cryptographic primitives such as hash functions, encryption, or digital signatures.

This lack of restriction on the number of Byzantine peers is a key aspect of our system model, as it reflects the dynamic nature of peer-to-peer systems where peers can join or leave at will. It distinguishes our model from Byzantine fault-tolerant consensus

systems, which often assume that Byzantine peers constitute less than a third of the total peers.

We assume, without loss of generality, that Byzantine peers cannot impersonate other peers or alter the content of messages exchanged between them. This is ensured by cryptographic techniques, such as digital signatures, which guarantee message authenticity and integrity among correct peers. This assumption does not limit the model's applicability, as these security measures are commonly implementable in practical peer-to-peer systems.

Beyond this limitation, Byzantine peers can engage in any arbitrary behavior, including but not limited to malicious actions designed to disrupt the system or gain unfair advantages. Some examples of such behaviors include:

- Pretend to be offline by not responding to messages or selectively ignoring certain messages.
- Send corrupted messages that do not conform to the expected format or protocol.
- Send contradictory messages to different peers, causing inconsistencies in the system, which is known as *equivocation*.
- Generate and propagate malicious messages designed to disrupt the system.

These examples demonstrate possible behaviors of Byzantine peers, though they are not exhaustive. Due to the unpredictable and potentially malicious nature of Byzantine behavior, many other actions are possible. Such peers may devise new attack strategies or combine multiple tactics in unexpected ways.

The unpredictability of accounting for all possible Byzantine behaviors highlights the importance of formal verification in designing Byzantine fault-tolerant systems. Isabelle/HOL offers a mathematical framework to formally verify the system properties in the face of arbitrary adversarial actions. This enables the development of protocols that can resist both known and unforeseen attacks.

4.3 Correctness Model

Our system's correctness is based on the Strong Eventual Consistency (SEC) model, the standard model of correctness for CRDTs. In the context of Byzantine fault tolerance, a system is considered correct if and only if it can maintain SEC among correct peers, regardless of the presence and actions of any number of Byzantine peers.

Specifically, our correctness model requires the following properties to hold:

- **Eventual Delivery:** Any update made by a correct peer is eventually delivered to all other correct peers, despite the presence of Byzantine peers.
- **Strong Convergence:** All correct peers that have processed the same set of updates are in an equivalent state, regardless of the order in which these updates were received or any interference from Byzantine peers.
- **Termination:** All operations initiated by correct peers eventually complete, even in the presence of Byzantine behavior.

These properties only apply to correct peers, while Byzantine peers may diverge or reach inconsistent states, but this does not affect the system's overall correctness as long as all correct peers maintain SEC, since by definition it is not possible to make any guarantees about the internal state of Byzantine peers.

5 Known Attacks on CRDTs

In this chapter, we will discuss the known attacks on CRDTs. We will first introduce the eventual delivery attack, which is a type of attack that aims to prevent the correct peers from delivering the same set of updates. Then, we will introduce the CRDT-specific attacks, which are attacks that aim to exploit the vulnerability of the specific CRDTs, resulting in convergence failure, even if the correct peers deliver the same set of updates.

5.1 Eventual Delivery Attack

The eventual delivery attack [KH20] aims to prevent correct peers from delivering the same set of updates, thereby violating the eventual delivery property of CRDTs. In theory, if communication cost is not a concern, achieving eventual delivery in a Byzantine environment is relatively straightforward. A simple solution would be:

- When two correct peers communicate, they periodically exchange updates that they have not received from each other.
- This process continues until both peers have received all updates from each other.
- This exchange happens between all pairs of correct peers in the network.
- As a result, updates propagate through the network, ensuring that all correct peers eventually receive all updates.

This naive approach guarantees eventual delivery because every update originating from a correct peer will be continuously forwarded until it reaches all other correct peers. Byzantine peers cannot prevent the propagation of updates between correct peers, as all the correct peers are directly or indirectly connected and Byzantine peers cannot alter the message content between correct peers. Message loss or delay is no problem because updates are periodically resent until received from the other peer.

However, this solution is impractical in real-world scenarios due to its high communication overhead. For example, if peers p and q each receive an update from peer r, they must both send r's update to each other to ensure the other has received it, even though this is redundant. Therefore, in practice, systems generally employ mechanisms like *version vectors* [Par+83] to track and exchange updates more efficiently.

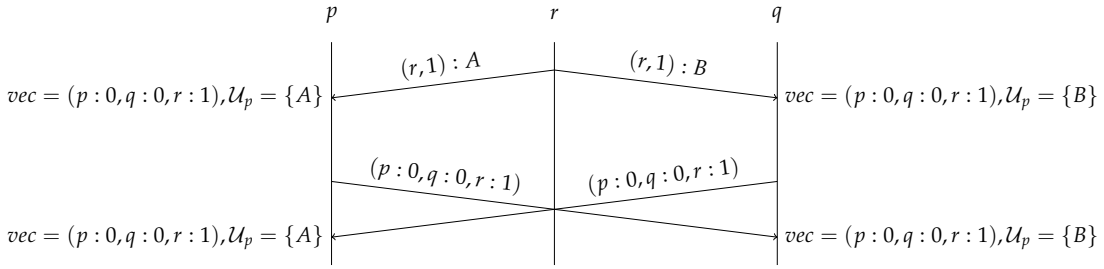

Figure 5.1: Byzantine peer r sends different updates A and B to p and q, using the same ID (r, 1). This causes p and q to falsely believe they have identical update sets, despite having different $\mathcal{U}_p$ and $\mathcal{U}_q$.

Version vectors are a mechanism used in distributed systems to track the state of updates across different peers. Each peer maintains a vector of counters, with one counter for each peer in the system. When a peer makes an update, it increments its own counter in its version vector. When peers communicate, they exchange their version vectors to determine which updates each peer has seen.

This approach allows peers to efficiently determine which updates they are missing without having to send the updates they already share. However, this efficiency comes at a cost: it introduces a potential vulnerability that Byzantine peers can exploit.

Figure 5.1 illustrates the vulnerability of version vectors to the eventual delivery attack. In this scenario, a Byzantine peer r sends two different updates, A and B, to correct peers p and q respectively, while deceptively using the same identifier (r, 1) for both. Consequently, p and q erroneously believe they have delivered the same set of updates, as their version vectors are identical, even though they have different $\mathcal{U}_p$ and $\mathcal{U}_q$.

5.2 CRDT-specific Attacks

This section focuses on attacks targeting the vulnerabilities of specific CRDT algorithms. These vulnerabilities often stem from the assumption that all peers will adhere to the algorithm's rules when generating and broadcasting updates. In a typical CRDT implementation, each peer is expected to perform checks to ensure its updates comply with these rules. However, Byzantine peers can circumvent these safeguards by directly broadcasting invalid updates. Receiving nodes may not be able to detect that the update is invalid. Ultimately, this can lead to a situation where a peer, upon receiving an invalid update, may directly apply it, resulting in unexpected errors.

To address these vulnerabilities, a more robust approach is to shift the responsibility

of validity checks from the sender to the receiver. Instead of relying on each peer to validate its own updates before broadcasting, we can implement a system where every correct peer performs a validity check on incoming updates before applying them.

This approach, however, introduces a new risk: Byzantine peers can exploit the validity check to cause inconsistency among different peers regarding the validity of an update. In other words, even if two peers deliver the same set of updates, their differing views on validity can lead them to apply different sets of updates, thereby preventing two correct peers from converging.

To illustrate how Byzantine peers can exploit validity checks to cause divergence among correct peers, we will examine two common CRDT types: *ORSet* and *RGA*. In both cases, we will demonstrate how a Byzantine peer can craft updates that pass validity checks on some correct peers but fail on others, leading to inconsistent states.

5.2.1 ORSet

Attack: Referencing Non-existent ID

A remove operation is only considered valid if the identifier it's trying to remove actually exists in the set. If it is not, the operation can be simply ignored. However, in a Byzantine environment, this check can be exploited to cause divergence among correct peers.

Consider the following scenario, illustrated in Figure 5.2:

1. A Byzantine peer r sends two concurrent updates to correct peers p and q.
2. To peer p, r sends an Add operation: $(Add, \text{"Hello"}, id_1)$.
3. To peer q, r sends a Remove operation: $(Rem, \text{"Hello"}, [id_1])$.
4. Later, r sends the Remove operation to p and the Add operation to q.

In this scenario, peer p first adds "Hello" with id_1, and then removes it when it receives the Remove operation. The final state of p is an empty set.

However, peer q first receives the Remove operation. Since "Hello" with id_1 doesn't exist in q's state yet, this Remove operation is considered invalid and is ignored. When q later receives the Add operation, it adds "Hello" with id_1 to its set.

As a result, even though both p and q have received the same set of updates, their final states diverge: p's set is empty, while q's set contains "Hello". This divergence occurs because the validity of the Remove operation depends on the order in which updates are received, which can be manipulated by a Byzantine peer.

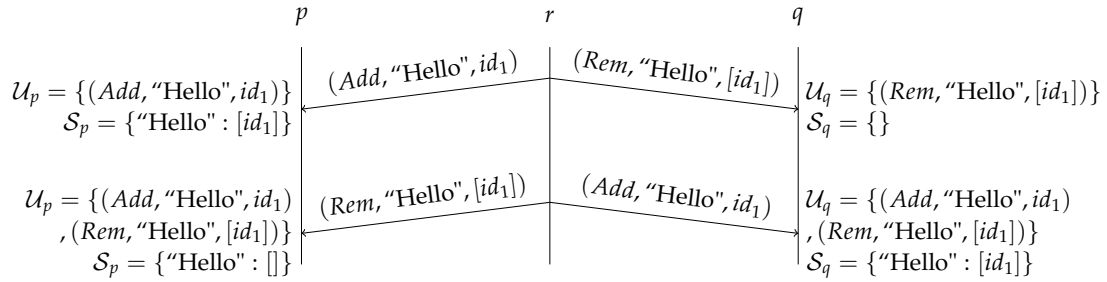

Figure 5.2: Byzantine peer r sends two concurrent updates to p and q in different order. The Add operation is applied to p, followed by the Remove operation, resulting in an empty set. However, q applies the Remove operation first, which is invalid and ignored by p. Followed by the Add operation, the state becomes $\{\text{“Hello"} : [id_1]\}$, leading to divergence.

In a non-Byzantine environment, each peer performs the validity check locally before broadcasting an update. Therefore, a causal broadcast mechanism can prevent the receiver from delivering the Remove operation before the Add operation, thus avoiding the divergence.

5.2.2 RGA

RGA implements two validity checks for each operation:

1. When receiving an Add operation, it verifies whether the ID already exists to prevent duplicate IDs.
2. For both Add and Remove operations that reference another ID, it checks if the referenced ID exists. If not, the operation is ignored.

RGA is vulnerable to two main types of Byzantine attacks:

Attack 1: Conflicting Inserts with Same ID

This attack exploits the uniqueness of operation IDs:

1. A Byzantine peer r sends two conflicting Insert operations with the same ID but different elements to two correct peers p and q.
2. Peer p receives $(Insert, (A, id_1), None)$, while q receives $(Insert, (B, id_1), None)$.

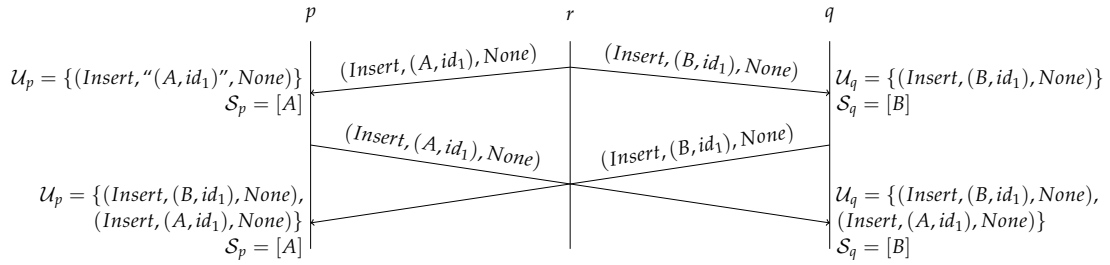

Figure 5.3: Byzantine peer r sends two Insert operations to p and q with the same ID but different elements A and B. p and q apply the operation from r first, resulting in [A] and [B] respectively. Then, p and q exchange their updates. Since the ID is already present in the history, both p and q will consider the update as invalid and ignore it, leading to divergence.

3. Both p and q apply these operations, resulting in different states: $[A]$ for p and $[B]$ for q.
4. When p and q exchange their updates, both ignore the other's update because the ID already exists in their history.

This attack leads to a permanent divergence between p and q, violating the Strong Convergence property of SEC. Figure 5.3 illustrates this attack.

Attack 2: Exploiting Reference Checking

This attack takes advantage of the reference checking mechanism:

1. A Byzantine peer r sends an Insert operation referencing a non-existent ID id_1 to peer q: $(Insert, (B, id_2), id_1)$.
2. Concurrently, r sends a valid Insert operation with ID id_1 to p: $(Insert, (Aid_1), None)$.
3. Peer q ignores the operation due to the non-existent reference, while p applies it.
4. Peer r sends $(Insert, (B, id_2), id_1)$ to q and $(Insert, (A, id_1), None)$ to p. Both operations are valid and are applied by p and q respectively.

This attack results in q having $[A]$, while p has $[A, B]$, again violating Strong Convergence. Figure 5.4 demonstrates this scenario.

Both attacks highlight RGA's vulnerability to Byzantine behavior, emphasizing the need for additional mechanisms to ensure consistency in adversarial environments.

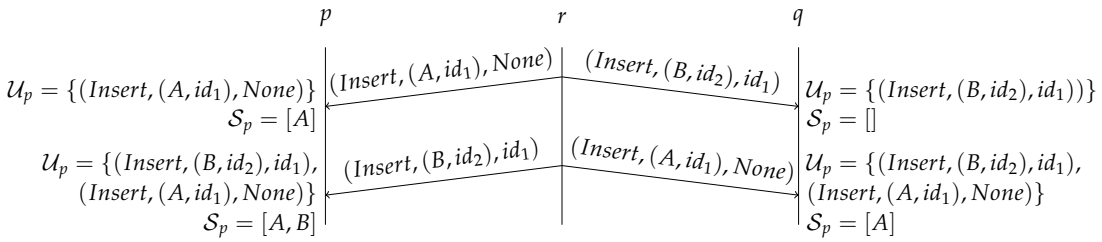

Figure 5.4: Byzantine peer r sends two concurrent Insert operations to p and q where one operation references the ID of the other operation. p applies the Insert A operation first, followed by the Insert B operation, resulting in [A, B]. q applies the Insert B operation first, which is an invalid operation as it references an ID that has not been added to the history yet, resulting in []. q then applies the Insert A operation, resulting in [A]. As a result, p and q diverge.

6 System Design and Formalization

6.1 System Overview

A Hash DAG is a directed acyclic graph (DAG) in which each node contains a value and a set of predecessor hashes that resolve to predecessor nodes in the hash graph. In our model, each peer maintains a hash graph, and the value of each node is a CRDT operation generated by some peer. The ancestor relationship between these nodes represents the causal relationship between the CRDT operations. This approach captures the causal dependencies between operations without the need of a causal broadcast network protocol.

When a peer generates a new hash node, it ensures that the CRDT operation contained in the new node is causally after all operations the peer has seen so far. When a peer receives a hash node from a remote peer, it waits until all predecessors of the received hash node are present in its local hash graph, and then adds the new hash node to its hash graph.

When a hash node is added to the hash graph, the CRDT operation contained is delivered. Therefore, a CRDT operation is only delivered after all its causally preceding operations have been delivered.

This approach provides several benefits:

1. Causal Delivery: It guarantees that operations are delivered in a causally consistent order across all correct peers.

2. Byzantine Eventual Delivery: The use of hash graphs makes it difficult for Byzantine peers to prevent the eventual delivery of updates to correct peers. We will formally verify this property in Section 6.2.

3. Consistent Validity Check: If the validity check only depends on the ancestors of the incoming node, we can perform the validity check consistently across all peers, thus avoiding the vulnerabilities outlined in Section 5.2. This property is formally verified in Section 6.3.4.

6.2 Hash DAG

We use a Hash DAG as the underlying data structure for various purposes, including tracking the causal relationships between operations, ensuring the eventual delivery of updates, and providing a foundation for validity checks of incoming updates. This construction was informally proposed by Kleppmann [Kle22]. We now show how to formalize the definition of a hash graph, and prove its properties that we will need later.

6.2.1 Definition of Hash Graph

A hash node is a pair of a finite set of hashes and a value. The set of hashes represent the predecessors of the node in the graph, and the value is the value associated with this node, e.g a CRDT operation. A hash graph is a finite set of hash nodes. In Isabelle, operators for finite sets need to be enclosed in vertical bars, but for simplicity, we use the usual set notation in this thesis.

type-synonym $('hash, 'val)$ *node* $=$ ‹$('hash\ fset \times 'val)$›
type-synonym $('hash, 'val)$ *hash-graph* $=$ ‹$('hash, 'val)$ *node fset*›

We define the locale of the hash graph, assuming the hash function is collision-free. Although, in theory, hash functions may have collisions, with cryptographic hash functions such as SHA-3 it is generally believed to be computationally infeasible to find them with better than negligible probability. Therefore, for the purposes of our formalization, we can safely assume that collisions do not occur. In the following locale, we fix a hash function H that satisfies the collision-free property.

type-synonym $('hash, 'val)$ *hash-func* $=$ ‹$('hash, 'val)$ *node* $\Rightarrow$ $'hash$›
locale *hash-graph* $=$
  fixes H :: ‹$('hash, 'val)$ *hash-func*›
  assumes *hash-no-collisions* : ‹$(\forall\, x\ y.\ H\ x = H\ y \longrightarrow x = y)$›

6.2.2 Heads, Predecessors and Ancestors

We define the *head-nodes* of a hash graph, which is the set of nodes that have no successors in the graph. *FSet.ffilter* uses a predicate to filter the set of hash nodes, in this case that the node has no successors. *heads* is then the image of this set under the hash function H. In Isabelle ' is the infix operator for the image of a set under a function.

definition *head-nodes* :: ‹$('hash, 'val)$ *hash-graph* $\Rightarrow$ $('hash, 'val)$ *node fset*› **where**
  ‹*head-nodes G* $=$*FSet.ffilter* $(\lambda n.\ \neg(\exists\,(hashes, \text{-}) \in G.\ H\ n \in hashes))$ *G*›
definition *heads* :: ‹$('hash, 'val)$ *hash-graph* $\Rightarrow$ $'hash$ *fset*› **where**
  ‹*heads G* $= H$ ' (*head-nodes G*)›

We then define the predecessors of a node as the set of nodes that have the given node as a successor.

definition *preds-of-node* :: ‹('hash, 'val) node ⇒ ('hash, 'val) node set› **where**
‹*preds-of-node node* ≡ {*x*. *H x* ∈ *fst node*}›

The ancestor relation is defined inductively so that it is the transitive closure of the predecessor relation. We also define the reachable relation as the reflexive transitive closure of the predecessor relation. The infix operator $\prec$ is used for the ancestor relation, and $\preceq$ is used for the reachable relation.

inductive *ancestor* :: ‹('hash, 'val) node ⇒('hash, 'val) node ⇒ bool› (**infix** ‹≺› 50) **where**
pred: ‹*H a* ∈ *fst b* ⟹ *a* ≺ *b*›
tran: ‹⟦*a* ≺ *b*; *b* ≺ *c*⟧ ⟹ *a* ≺ *c*›
definition *reachable* :: ‹('hash, 'val) node ⇒('hash, 'val) node ⇒ bool› (**infix** ‹⪯› 50) **where**
‹*a* ⪯ *b* ≡ *a* ≺ *b* ∨ *a* = *b*›

We then define the set of ancestors of a node and the set of reachable nodes of a node as follows:

definition *ancestor-nodes-of* :: ‹('hash, 'val) node ⇒ ('hash, 'val) node set› **where**
‹*ancestor-nodes-of node* ≡ {*x*. *x* ≺ *node*}›
definition *reachable-nodes-of* :: ‹('hash, 'val) node ⇒ ('hash, 'val) node set› **where**
‹*reachable-nodes-of node* ≡ {*x*. *x* ⪯ *node*}›

Finally we define the reachable nodes of a set of nodes as the union of the reachable nodes of each node in the set.

definition *reachable-nodes-of-set* :: ‹('hash, 'val) node set ⇒ ('hash, 'val) node set› **where**
‹*reachable-nodes-of-set nodes* ≡ ⋃ (*reachable-nodes-of* ' *nodes*)›

6.2.3 Structural Validity

A hash graph is defined to be structurally valid if it is either empty, or it is obtained by extending a valid hash graph by a new hash node that satisfies the following conditions, which can easily be enforced in practice:

- All the hashes in the new node resolve to nodes that are already in the graph, i.e the predecessors of the new node are in the graph.
- The new node itself is not in the graph.

- The hash of the new node is not in the set of hashes of the new node, i.e it doesn't reference itself. Constructing a node that references itself is similar to constructing a hash collision, and is also believed to be computationally infeasible. For the sake of proof, it is convenient to make this a separate assumption.

From the above description, we use an inductive predicate to define *struct-valid* as follows:

inductive *struct-valid* :: ‹(*'hash*, *'val*) *hash-graph* ⇒ *bool*› **where**
empty-graph: ‹*struct-valid* {}›
extend-graph: ‹⟦*struct-valid G*;
$\forall h \in hashes.\ \exists n \in G.\ H\ n = h$;
$(hashes, val) \notin G$;
$H\ (hashes, val) \notin hashes$
⟧ ⟹ *struct-valid* $(G \cup \{(hashes, val)\})$›

From the validity of the hash graph, we can easily prove the following lemma:

lemma *graph-hashes-closed*:
assumes ‹*struct-valid G*›
and ‹$(hashes, val) \in G$›
and ‹$h \in hashes$›
shows ‹$\exists n \in G.\ H\ n = h$›

Proof. By induction on the graph G:

For the base case, if G is empty, then the lemma holds trivially, as there is no node in G.

For the inductive step, if G is obtained by extending a valid hash graph G' by a new hash node *(hashes, val)*, then by the definition of *struct-valid*, all the hashes in the new node resolve to nodes that are already in the graph, i.e *hashes* is a subset of the nodes hashes in G'. Therefore, the lemma holds. □

Similarly, we can prove the following lemma that no node references itself:

lemma *node-not-referencing-itself*:
assumes ‹*struct-valid G*›
shows ‹$\forall (hashes, val) \in G.\ H\ (hashes, val) \notin hashes$›

Additionally, we can prove that old nodes do not reference new nodes.

lemma *old-nodes-not-referencing-new-node*:
assumes ‹*struct-valid G*›
and ‹$node \notin G$›
and ‹*struct-valid* $(G \cup \{node\})$›
shows ‹$\forall (hs, val) \in G.\ H\ node \notin hs$›

Proof. The proof is by contradiction. We assume an old node references a new node, and we can conclude that there exists a hash node *(hashes, val)* in G such that H *node* $\in$ *hashes*. This contradicts the fact that all the hashes in G are already in the graph using lemma *graph-hashes-closed*, i.e H *node* $\notin$ *hashes*. □

6.2.4 Properties of Head Nodes

We first prove that when a node is added to a structurally valid graph, the new node is always a head node in the new graph:

lemma *new-node-is-head-node*:
assumes ‹*struct-valid G*›
and ‹$(hashes, val) \notin G$›
and ‹*struct-valid* $(G \cup \{(hashes, val)\})$›
shows ‹$(hashes, val) \in$ *head-nodes* $(G \cup \{(hashes, val)\})$›

Proof. From the assumptions and lemma *old-nodes-not-referencing-new-node*, we can conclude that there does not exist a hash node in G that references the new node. Moreover, from lemma *node-not-referencing-itself*, we know that the new node does not reference itself. Therefore, the new node is not referenced by any node in $G \cup$ *{(hashes, val)}*. Hence, the new node is a head node in the new graph.

□

We also prove that the head nodes of the new graph are the head nodes of the old graph minus the predecessors of the new node, and plus the new node itself. *head-nodes-set* convert the finite set version of *head-nodes* into normal set for Isabelle technical reasons.

lemma *head-nodes-update-on-new-node-addition*:
assumes ‹*struct-valid G*›
and ‹$(hashes, val) \notin G$›
and ‹*struct-valid* $(G \cup \{(hashes, val)\})$›
shows ‹*head-nodes-set* $(G \cup \{(hashes, val)\}) =$ *head-nodes-set G* $-$ *preds-of-node* $(hashes, val) \cup \{(hashes, val)\}$›

Proof. We first prove that the left hand side of the equation is a subset of the right hand side. Assume x is a head node in the new graph, i.e $x \in$ *head-nodes-set (G* $\cup$ *{(hashes, val)}*). We need to show that $x \in$ *head-nodes-set G* $-$ *preds-of-node (hashes, val)* $\cup$ $\{(hashes, val)\}$. We consider two cases:

- If $x =$ *(hashes, val)*, then x is in right hand side trivially.

- If $x \neq$ *(hashes, val)*, then x is not in the predecessors of the new new node, since predcessors of the new node must not be head nodes in the new graph. Moreover, x is not in $(G -$ *head-nodes-set* $G)$, since a non-head node in the old graph cannot be a head node in the new graph. Therefore, x is an element that is in the old head nodes set but not in the predecessors of the new node. Hence, $x \in$ *head-nodes-set* $G -$ *preds-of-node (hashes, val)* $\cup \{(hashes, val)\}$.

We then prove the right hand side is a subset of the left hand side. Assume $x \in$ *head-nodes-set* $G -$ *preds-of-node (hashes, val)* $\cup \{(hashes, val)\}$. We need to show that $x \in$ *head-nodes-set (*$G \cup$*{(hashes, val)})*. We consider two cases:

- If $x =$ *(hashes, val)*, then x is in the left hand side trivially, since new node is always a head node in the new graph.
- If $x \neq$ *(hashes, val)*, then x is in the old head nodes set but not in the predecessors of the new node, i.e $x \in$ *(head-nodes-set* $G -$ *preds-of-node (hashes, val))*. Those nodes don't have a successor in the new graph, so they are still head nodes in the old graph. Therefore, $x \in$ *(head-nodes-set (*$G \cup$*{(hashes, val)}))*.

□

6.2.5 Properties of Reachable Nodes

Firstly, we prove that the reachable nodes of a node n is the union of the reachable nodes of the predecessors of n and n itself.

lemma *reachable-nodes-preds-equiv*:
assumes ‹*ns* = *preds-of-node n*›
shows ‹*reachable-nodes-of-set ns* ∪ {*n*} = *reachable-nodes-of n*›

Proof. We first prove that the reachable nodes of a set of nodes ns plus the node n is a subset of the reachable nodes of the node n.

Assume $x \in$ *reachable-nodes-of-set* $ns \cup \{n\}$. We need to show that $x \in$ *reachable-nodes-of* n. We consider two cases:

- If $x = n$, then $x \in$ *reachable-nodes-of* n trivially, as $n \prec n$.
- If $x \neq n$, then $x \in$ *reachable-nodes-of-set* ns. By the definition of *reachable-nodes-of-set*, there exists a node $n' \in ns$ such that $x \preceq n'$. Since ns is the predecessors of n, and n' is a node in ns, we have $n' \prec n$. Therefore, by the transitivity of the reachable relation, we have $x \preceq n$. Therefore, $x \in$ *reachable-nodes-of* n.

We then prove that the reachable nodes of the node n is a subset of the reachable nodes of a set of nodes ns plus the node n.

Assume $x \in$ *reachable-nodes-of* n. We need to show that $x \in$ *reachable-nodes-of-set* $ns \cup \{n\}$. We consider two cases:

- If $x = n$, then $x \in$ *reachable-nodes-of-set* $ns \cup \{n\}$ trivially.
- If $x \neq n$, since $x \in$ *reachable-nodes-of* n, by the definition of *reachable-nodes-of*, there exists a predecessor node $n' \in$ *preds-of-node* n such that $x \preceq n'$. Since ns is the predecessors of n, and n' is a node in ns, we have $n' \prec n$. By transitivity, $x \preceq n$. Therefore, $x \in$ *reachable-nodes-of-set* ns.

□

We then prove the reachable nodes of a node n in graph G is subset of G.

lemma *reachable-of-G-node-in-G*:
assumes ‹*struct-valid G*›
and ‹$n \in$ *fset G*›
shows ‹*reachable-nodes-of n* $\subseteq$ *fset G*›

Proof. The proof is by induction on the graph G.

For the base case, if G is empty, then there isn't any node in G. The assumption is not true, hence the lemma holds.

For the inductive step, if G' is obtained by extending a valid graph G by a new node *(hashes, val)*, we consider two cases:

- If $n =$ *(hashes, val)*, then

  reachable-nodes-of n = reachable-nodes-of-set (preds-of-node (hashes, val)) ∪ {(hashes, val)}

  according to lemma *reachable-nodes-preds-equiv*. By the induction hypothesis, we know that *reachable-nodes-of-set (preds-of-node (hashes, val))* $\subseteq$ *fset G*. Therefore, *reachable-nodes-of n* $\subseteq$ *fset G'*.

- If $n \neq$ *(hashes, val)*, then by the induction hypothesis, we know that

  reachable-nodes-of n $\subseteq$ *fset G'*

□

By using the lemma above, we can easily conclude the following lemma:

lemma *reachable-nodes-of-heads-is-subset-of-graph*:
assumes ‹*struct-valid G*›
shows ‹*reachable-nodes-of-set* (*head-nodes-set G*) ⊆ *fset G*›

We then prove that when a new node is added, the reachable nodes of the heads of the old graph, plus the new node itself, form a subset of the reachable nodes of the heads of the new graph.

lemma *heads-subset-after-new-node-addition*:
assumes ‹*struct-valid G*›
and ‹(*hs*, *v*) ∉ *G*›
and ‹$G' = G \cup \{(hs, v)\}$›
and ‹*struct-valid* ($G \cup \{(hs, v)\}$)›
shows ‹(*reachable-nodes-of-set* (*head-nodes-set G*)) ∪ {(*hs*, *v*)}
⊆ *reachable-nodes-of-set* (*head-nodes-set G'*)›

Proof. Assume *x* ∈ *reachable-nodes-of-set (head-nodes-set G)* ∪ *{(hs, v)}*. We need to show that *x* ∈ *reachable-nodes-of-set (head-nodes-set G')*. We consider two cases:

- If *x* = *(hs, v)*, then *x* ∈ *reachable-nodes-of-set (head-nodes-set G')* trivially.
- If *x* ≠ *(hs, v)*, then *x* ∈ *reachable-nodes-of-set (head-nodes-set G)*. By the definition of *reachable-nodes-of-set*, there exists a head node *y* ∈ *head-nodes-set G* such that $x \preceq y$. We then consider two subcases:
  - If *y* ∈ *head-nodes-set G'*, then $x \preceq y$ implies the lemma holds by the definition of *reachable-nodes-of-set*.
  - If *y* ∉ *head-nodes-set G'*, then *y* ∈ *preds-of-node (hs, v)* according to lemma *head-nodes-update-on-new-node-addition*. Since $x \preceq y$ and $y \prec (hs, v)$, by transitivity, we have $x \prec (hs, v)$. Since (hs, v) is a head node in G', we can conclude that *x* ∈ *reachable-nodes-of-set (head-nodes-set G')*.

□

We then prove that the graph *G* is a subset of the reachable nodes of the heads of G'.

lemma *graph-is-subset-of-reachable-nodes-of-heads*:
assumes ‹*struct-valid G*›
shows ‹*fset G* ⊆ *reachable-nodes-of-set* (*head-nodes-set G*)›

Proof. By induction on the graph *G*.

For the base case, if *G* is empty, then the lemma holds trivially, since both sides of the subset relation are empty.

For the inductive step, we assume G' is obtained by extending a valid graph G by a new node *(hashes, val)*. From the induction hypothesis, we know that *fset G* $\subseteq$ *reachable-nodes-of-set (head-nodes-set G)*. If we can prove

$$\begin{aligned}&\textit{reachable-nodes-of-set(head-nodes-set G')} = \\ &\textit{(reachable-nodes-of-set (head-nodes-set G))} \cup \{(\text{hashes}, \text{val})\}\end{aligned}$$

then the lemma holds. In the following proof, we will show it by proving the left hand side is a subset of the right hand side and vice versa.

- From lemma *heads-subset-after-new-node-addition*, we can easily show the right hand side is a subset of the left hand side.
- To show the left hand side is a subset of the right hand side, we consider an arbitrary element x in the left hand side. We do a case analysis on whether $x =$ *(hashes, val)*.
  - If $x =$ *(hashes, val)*, then the lemma holds trivially.
  - If $x \neq$ *(hashes, val)*, from the definition of *reachable-nodes-of-set*, we can obtain a node $y \in$ *head-nodes-set G'* such that $x \preceq y$. We then consider two subcases:
    * If $y \neq$ *(hashes, val)*, then y is in head node set of G, so that we can conclude the lemma.
    * If $y =$ *(hashes, val)*, then we can obtain a node $z \in$ *preds-of-node (hashes, val)* such that $x \preceq z$. According to the induction hypothesis, z is in

      $$\textit{reachable-nodes-of-set (head-nodes-set G)}$$

      Since $x \preceq z$, by transitivity, we have

      $$x \in \textit{reachable-nodes-of-set (head-nodes-set G')}$$

□

lemma *reachable-nodes-of-heads-is-graph*:
assumes ‹*struct-valid G*›
shows ‹*reachable-nodes-of-set* (*head-nodes-set G*) = *fset G*›

Proof. Follows directly from *graph-is-subset-of-reachable-nodes-of-heads* and lemma *reachable-nodes-of-heads-is-subset-of-graph*.

□

We finally reach an important conclusion that if two graphs are structurally valid and share the same heads, the two must be equal. This is a widely-known fact about hash DAGs, but as this section showed, proving it is not straightforward.

theorem *heads-define-graph*:
assumes ‹*struct-valid G1*› **and** ‹*struct-valid G2*›
and ‹*heads G1 = heads G2*›
shows ‹*G1 = G2*›

Proof. Follows from the assumption that the hash function has no collisions, and lemma *reachable-nodes-of-heads-is-graph*.

□

This theorem shows that when two peers need to synchronize their updates, they can simply exchange the hashes of their head nodes. If these hashes are identical, the nodes can be confident that their entire hash graphs, including all updates contained within them, are exactly the same.

In cases where the head hashes don't match, the nodes can initiate a set reconciliation protocol [Wui+24; KH20; YGA24; SG21; Mey23], which allows them to efficiently exchange the missing elements of their respective hash graphs. Once this exchange is complete, they can re-validate their synchronization by comparing head hashes again.

This method offers a secure alternative to the version vector protocol discussed in Section 5.1. It is not susceptible to eventual delivery attacks. Each correct node can independently guarantee the structural validity of its hash graph. The only additional assumption required is that the hash function is collision-resistant, which can be achieved by employing a cryptographic hash function that Byzantine peers cannot break.

6.3 BFT Convergence

We model the system as a network of multiple peers, where each peer has a node history. This history represents a list of hash graph nodes that are either generated locally or received from remote peers. We assume that correct peers discard received messages that do not conform to the hash graph node format. Besides, we assume that the attacker cannot construct multiple hash graph nodes such that they form a cycle. This is a reasonable assumption because, for a cryptographic hash function, this is generally believed to be computationally infeasible. We place no other restrictions on the history. In other words, we make no assumptions about the attacker: an attacker can send any hash graph node to the victim, combine them arbitrarily, collaborate

with other attackers, and even control the order in which these hash graph nodes are received.

Additionally, the model requires a function *interp'* that represents an abstract interpretation function of a CRDT. It takes a hash graph node and transforms one state into another, similarly to Gomes et al. [Gom+17b]'s approach described in Section 3.2. The function *is-sem-valid'* represents the semantic validation operation of a CRDT. This means that any hash graph node in the history must pass the *is-sem-valid'* check before being interpreted. If it does not pass the check, it is discarded without being applied to the state. The definitions of *interp'* and *is-sem-valid'* are different for each CRDT. The *initial-state*, as the name suggests, is the initial CRDT state of each peer, which is also determined by the specific CRDT.

locale *peers-with-arbitrary-history* = *hash-graph H*
  for *H* :: ‹('*hash*, '*val*) *hash-func*› +
  fixes *history* :: ‹*nat* ⇒ ('*hash*, '*val*) *node list*›
    and *interp'* :: ‹('*hash*, '*val*) *hash-func* ⇒ ('*hash*, '*val*) *node* ⇒ '*state* ⇀ '*state*›
    and *initial-state* :: ‹'*state*›
    and *is-sem-valid'* :: ‹('*hash*, '*val*) *causal-func* ⇒ ('*hash*, '*val*) *hash-func* ⇒ ('*hash*, '*val*) *node set* ⇒ ('*hash*, '*val*) *node* ⇒ *bool*›
  assumes *no-cycle*: ‹$x \prec y \wedge y \prec x \Longrightarrow$ *False*›

To reuse the abstract convergence theory in Section 3.2.1, we instantiate the locale with ancestor relation. From this point, we can reuse all the abstract results in the locale *happens-before*.

sublocale *hb*: *happens-before reachable ancestor interp*
proof
  fix *x y*
  show ‹$(x \prec y) = (x \preceq y \wedge \neg\, y \preceq x)$›
    by (*simp add*: *ancestor-less-le-not-le*)
next
  fix *x y*
  show ‹$x \preceq x$›
    by (*simp add*: *reachable-def*)
next
  fix *x y z*
  show ‹$x \preceq y \Longrightarrow y \preceq z \Longrightarrow x \preceq z$›
    using *reachable-transitivity* **by** *blast*
qed

6.3.1 Behavior of Correct Peers

Each correct peer maintains a local state, which includes a list of delivered nodes and a hash graph. The delivered nodes refers to the sequence of hash graph nodes in the peer's history that have passed the structual and semantic validity checks. In practice, a peer only needs to store the hash graph, while the delivered nodes is used solely for proof purposes, as any delivered node is applied immediately to the CRDT state.

type-synonym *('hash, 'val) peer-state* = ‹*('hash, 'val) node list* × *('hash, 'val) hash-graph*›

A correct peer first checks both the semantic and structural validity of the node against the local state. If the node is valid, the peer delivers the update and add it to its hash graph.

```
fun is-valid :: ‹('hash, 'val) hash-graph ⇒ ('hash, 'val) node ⇒ bool› where
  ‹is-valid G x = ((is-sem-valid G x) ∧ (is-struct-valid G x))›
fun check-and-apply :: ‹('hash, 'val) peer-state ⇒ ('hash, 'val) node ⇒ ('hash, 'val) peer-state› where
  ‹check-and-apply (ah, G) n =
    (if is-valid G n then
      (ah @ [n], G ∪ {n})
    else
      (ah, G)
    )
›
```

To obtain the final state of a correct peer, we define a function that folds the history into the initial state, where both the initial state and the delivered nodes are empty. We reuse the *apply-operations* function from the abstract convergence theory to get the final CRDT state of a peer.

```
fun apply-history :: ‹('hash, 'val) peer-state ⇒ ('hash, 'val) node list ⇒ ('hash, 'val) peer-state› where
  ‹apply-history S ns = foldl check-and-apply S ns›
fun peer-state :: ‹nat ⇒ ('hash, 'val) peer-state› where
  ‹peer-state i = apply-history ([], {}) (history i)›
definition apply-operations :: ‹('hash, 'val) node list ⇒ 'state option› where
  ‹apply-operations ns ≡ hb.apply-operations ns initial-state›
```

We define two helpers to get the hash graph and the delivered nodes from the peer state.

```
definition graph :: ‹nat ⇒ ('hash, 'val) hash-graph› where
  ‹graph i = snd (peer-state i)›
definition delivered-nodes :: ‹nat ⇒ ('hash, 'val) node list› where
  ‹delivered-nodes i = fst (peer-state i)›
```

We then inductively define a predicate *sem-valid* that checks if a hash graph is semantically valid. A hash graph is *sem-valid* either if it is empty or if it is extended by a new hash node that is semantically valid with respect to a *sem-valid* hash graph.

inductive *sem-valid* :: ‹('*hash*, '*val*) *hash-graph* ⇒ *bool*› **where**
‹*sem-valid* {}›
‹⟦ *sem-valid G*; *is-sem-valid G n*⟧ ⟹ *sem-valid* (*G* ∪ {*n*})›

We further define a predicate *valid-graph* that checks if a hash graph is both structurally valid and semantically valid. Similarly, a hash graph is *valid-graph* if it is empty or it is extended by a new hash node that is both structurally valid and semantically valid with respect to a *valid-graph* hash graph.

inductive *valid-graph* :: ‹('*hash*, '*val*) *hash-graph* ⇒ *bool*› **where**
‹*valid-graph* {}›
‹⟦ *valid-graph G*; *is-valid G n*⟧ ⟹ *valid-graph* (*G* ∪ {*n*})›

6.3.2 Properties of a Correct Peer

For a correct peer, the hash nodes in the delivered nodes are always equal to the hash nodes in the local graph. This lemma can be easily shown by induction on the history of a correct peer.

lemma *peer-apply-history-preserve-same-nodes*:
assumes ‹*apply-history* ([], {}) *ns* = (*ah*, *G*)›
shows ‹*fset-of-list ah* = *G*›

Validity

We then show that, for a correct peer, the local graph is always a *valid-graph*.

lemma *apply-history-preserve-validity*:
assumes ‹*apply-history* (*ah*, *G*) *ns* = (*ah'*, *G'*)›
and ‹*valid-graph G*›
shows ‹*valid-graph G'*›

Proof. The proof is by induction on the history of a peer.

For the base case, the history is empty, thus the local graph is empty and it is trivially a *valid-graph*.

For the inductive step, before a new hash node is applied, the structual validlity and semantic validity are satisfied. Therefore, the new hash graph is also a *valid-graph*. □

Distinctness

We next show that the items in the delivered nodes of a correct peer are all distinct, i.e that is does not contain the same item more than once.

lemma *apply-history-preserve-distinctness*:
assumes ‹*apply-history* (*ah*, *G*) *ns* = (*ah'*, *G'*)›
and ‹*fset-of-list ah* = *G*›
and ‹*distinct ah*›
shows ‹*distinct ah'*›

Proof. The proof is by induction on the history of a peer.

For the base case, the history is empty, thus the local graph is empty and it is trivially *distinct*.

For the inductive step, before a new node is applied, the structrual validity is checked. Since structual validity requires the new node not to be presented in the old graph, the new node is not in the old delivered nodes, hence the lemma holds. □

Causal consistency

The delivered nodes of a correct peer is also consistent causally consistent as described in Section 3.2.1. That is, the ancestors of a hash node must appear before that node in the delivered nodes.

lemma *valid-history-causality-hold1*:
‹*valid-graph G* ⟹ *is-valid G x* ⟹ ∀ *y*. *y* ≺ *x* ⟶ *y* ∈ *fset G*›

Proof. Given a valid hash graph, for a structual valid hash node, all the ancestors of the new node must be in the graph by lemma *reachable-of-G-node-in-G*, hence the lemma holds. □

From lemma *valid-history-causality-hold1*, we can further conclude

lemma *valid-history-causality-hold2*:
‹*valid-graph G* ⟹ *is-valid G x* ⟹ ∀ *y* ∈ *fset G* . *y* ≺ *x* ∨ *y* ∥ *x*›

Finally, we can prove a peer's delivered nodes is always *hb-consistent*.

lemma *peer-history-hb-consistent*:
assumes ‹*apply-history* ([], {}) *ns* = (*ah*, *G*)›
shows ‹*hb.hb-consistent ah*›

Proof. The proof is by induction on the history of a peer.

For the base case, the history is empty, thus the delivered nodes is empty and the lemma holds trivially.

For the inductive case, the new node is applied to the history, thus the new delivered nodes is the old delivered nodes with the new node added. By lemma *valid-history-causality-hold2*, we know that all the old nodes are either concurrent or causally before the new node, hence the lemma holds.

□

6.3.3 Convergence

We can now prove the convergence theorem, which adds Byzantine fault tolerance to the abstract convergence theorem from Section 3.2.1. The theorem states that for two correct peers, if all concurrent operations commute, and the heads of the graphs of the two peers are equal, they reach the same final CRDT state.

lemma *convergence*:
assumes ‹*hb.concurrent-ops-commute* (*delivered-nodes i*)›
and ‹*hb.concurrent-ops-commute* (*delivered-nodes j*)›
and ‹*heads* (*graph i*) = *heads* (*graph j*)›
shows ‹*apply-operations* (*delivered-nodes i*) = *apply-operations* (*delivered-nodes j*)›

Proof. From the lemma *apply-history-preserve-validity*, we can conclude that the hash graphs of the two correct peers *i* and *j* satisfy the *valid-graph* predicate, hence the graphs are also *struct-valid*. From theorem *heads-define-graph* and the assumption that *i* and *j* share the same heads, *graph i* and *graph j* are equal.

From the lemma *peer-apply-history-preserve-same-nodes*, we know that the set of the applied history of *i* and *j* are the same, although the order might be different.

From the theorem *abstract-convergence*, lemma *apply-history-preserve-distinctness* and assumption that the operation are *concurrent-ops-commute*, we can further conclude that the final states of the two peers are the same.

□

The assumption *concurrent-ops-commute* is guaranteed by the CRDT algorithm, which we will prove later for different CRDTs.

6.3.4 Synchronization

While we have proven the convergence theorem, which states that two correct peers reach the same state if the CRDT algorithm ensures *concurrent-ops-commute* and the peers have the same heads, an attacker could potentially prevent two correct peers from

having the same heads. We need to prove that two correct peers can always synchronize to the same heads.

The synchronization process involves the following steps:

- Peers periodically exchange their current heads.
- When a peer receives heads from another peer, it compares them with its own.
- If there are differences, the peer requests the missing nodes from the other peer. Previous work [Wui+24; KH20; YGA24; SG21; Mey23] has proposed optimized algorithms to quickly identify the nodes that are missing from each peer's graph and efficiently transmit only those nodes, but for the purpose of this proof any reconciliation is sufficient.
- Upon receiving the missing nodes, the peer validates them and incorporates them into its own graph.

Since we assume the attacker cannot prevent correct peers from communicating, exchanging the missing nodes from each other is guaranteed to succeed. However, the attacker could attempt to construct a hash node that passes the validity check of node i, but fails the validity check of node j, thus preventing node j from adding the hash node to its graph.

More specifically, the attacker cannot prevent correct peers from failing the structural validity check. This is because structural validity only requires that a node's predecessors exist before it is added to the graph, the same node is not added more than once, and a node does not reference the hash of itself (which is computationally infeasible). As long as missing nodes are added in an order that is consistent with the *happens-before* relation, structural validity is guaranteed.

The real issue lies in semantic validity. It is conceivable that an attacker could construct a node that passes the semantic validity check on peer i but fails it on peer j.

To address this concern, we need to ensure that semantic validity checks are consistent across all correct peers. That is, two correct peers always make the same decision on whether or not a node is valid. This leads us to an important property of our BFT system: the semantic validity of a node must be deterministic and based solely on the information contained within the node and its ancestors in the hash graph. This criterion was proposed informally by Kleppmann [Kle22], and we now formalise it in our proof framework and show how this affects the synchronization process.

We capture this property by an assumption that only ancestors are relevant to semantic validity: the validity of node n does not change when any nodes that are concurrent with n are added to the graph.

CRDT algorithms that use a semantic validity check must then prove that it satifies this property.

definition *only-ancestors-relevant* :: ‹*bool*› **where**
‹*only-ancestors-relevant* = (∀ *n G*. (*ancestor-nodes-of n*) ⊆ *fset G* ⟶
is-sem-valid-set (*ancestor-nodes-of n*) *n* ⟷ *is-sem-valid G n*)›

6.3.5 Definition of valid sequence

To formalize the process of validating a sequence of nodes, we define the *valid-seq* function. This function takes three arguments:

1. A validity check function f
2. A sequence of nodes
3. A graph

The *valid-seq* function iterates through the sequence of nodes, applying the validity check function f to each node in the context of the current graph. If all nodes in the sequence pass the validity check, the function returns true. Otherwise, it returns false as soon as it encounters an invalid node.

fun *valid-seq* :: ‹((*'hash*, *'val*) *hash-graph* ⇒ (*'hash*, *'val*) *node* ⇒ *bool*)
⇒ (*'hash*, *'val*) *hash-graph* ⇒ (*'hash*, *'val*) *node list* ⇒ *bool*› **where**
‹*valid-seq f G* [] = *True*›
‹*valid-seq f G* (*x*#*xs*) =
(*if f G x then*
valid-seq f (*G*∪{*x*}) *xs*
else
False
)›

The *struct-valid-seq* and *sem-valid-seq* functions are helper functions that use *valid-seq* with *is-struct-valid* and *is-sem-valid* respectively as the validity check function.

definition *struct-valid-seq* :: ‹(*'hash*, *'val*) *hash-graph* ⇒ (*'hash*, *'val*) *node list* ⇒ *bool*› **where**
‹*struct-valid-seq G ns* ≡ *valid-seq is-struct-valid G ns*›
definition *sem-valid-seq* :: ‹(*'hash*, *'val*) *hash-graph* ⇒ (*'hash*, *'val*) *node list* ⇒ *bool*› **where**
‹*sem-valid-seq G ns* ≡ *valid-seq is-sem-valid G ns*›

6.3.6 Synchronization Proof

We firstly show that if a node is in a hash graph *G* that is *valid-graph*, it must be *sem-valid* with respect to *G*.

lemma *sem-valid-preserves*:
assumes ‹*valid-graph G*›
and ‹*n* ∈ *G*›
and ‹*only-ancestors-relevant*›
shows ‹*is-sem-valid G n*›

We then prove that if a node belongs to a valid graph G1, and it is structurally valid with respect to another valid graph G2, then it must also be semantically valid with respect to G2. The assumption that only ancestors are relevant for semantic validity is implicit in this lemma, since it is proved in the context of the *ancestor-only-check* locale defined previously.

lemma *sem-valid-consistent*:
assumes ‹*valid-graph G1*›
and ‹*valid-graph G2*›
and ‹*n* ∈ *G2*›
and ‹*is-struct-valid G1 n*›
and ‹*only-ancestors-relevant*›
shows ‹*is-sem-valid G1 n*›

Proof. From lemma *sem-valid-preserves*, we can conclude that *n* is semantically valid with respect to *G2*. Since *n* is in *G2*, we can know that all the ancestors must be in the *G2*. From the assumption that only ancestors are relevant for semantic validity, we can know that *n* is also semantically valid against the ancestors of *n*.

From the assumption that *n* is structually valid with respect to *G1*, the ancestors of *n* must also be in the *G1*. Using the assumption that only ancestors are relevant for semantic validity, we can conclude that *n* is also semantically valid with respect to *G1*. □

Based on the previous lemmas, we can now prove an important property about the validity of sequences of nodes across different valid graphs. This lemma states that if we have two valid graphs G1 and G2, and a sequence of nodes xs that belongs to G2, then if this sequence is structurally valid with respect to G1, it must also be semantically valid with respect to G1.

lemma *struct-valid-seq-implies-sem-valid-seq*:
assumes ‹*valid-graph G1*›

and ‹*valid-graph G2*›
and ‹$\forall n \in set\ xs.\ n \in G2$›
and ‹*struct-valid-seq G1 xs*›
and ‹*only-ancestors-relevant*›
shows ‹*sem-valid-seq G1 xs*›

Proof. The proof is by induction on *xs*.

For the base case, the sequence is empty, thus it is trivially valid.

For the inductive step, we know that *xs* is both structurally valid and semantically valid with respect to *G1*. By induction hypothesis, we know that *xs @ [x]* is structurally valid with respect to *G1*, thus *x* is structurally valid with respect to $G1 \cup fset\text{-}of\text{-}list\ xs$. Since *x* is in *G2*, it is also semantically valid with respect to $G1 \cup fset\text{-}of\text{-}list\ xs$ according to lemma *sem-valid-preserves*. Finally, *xs @ [x]* is both structurally valid and semantically valid with respect to *G1*.

□

This lemma bridges the gap between structural and semantic validity across different valid graphs. It ensures that if nodes from one valid graph can be structurally integrated into another valid graph, they will also be semantically valid with that graph.

A corollary of the above lemma is that if we have two valid graphs G1 and G2, and a sequence of nodes xs that are missing from G1 but present in G2, then if xs is structurally valid with respect to G1, it must also be semantically valid with respect to G1.

lemma *synchronization*:
assumes ‹*only-ancestors-relevant*›
and ‹*valid-graph G1*›
and ‹*valid-graph G2*›
and ‹$fset\text{-}of\text{-}list\ xs = G2 - G1$›
and ‹*struct-valid-seq G1 xs*›
shows ‹*sem-valid-seq G1 xs*›

Consider two correct peers, 1 and 2, with their respective graphs G1 and G2. By lemma *apply-history-preserve-validity*, both G1 and G2 are valid graphs. The sequence *xs* represents the nodes that are present in G2 but missing from G1. The theorem states that if *xs* can be applied to G1 in a structurally valid order, then *xs* is guaranteed to be semantically valid with respect to G1. This means that peer 1 can apply all the missing nodes from peer 2's graph to its own graph. As the theorem is symmetric, the same principle applies in reverse: peer 2 can apply all the nodes it's missing from peer 1's graph.

This leads to a conclusion: as long as semantic validity is only dependent on ancestors, we can guarantee that any two correct peers will be able to synchronize to identical

graphs, i.e Byzantine peers cannot prevent correct peers from synchronizing their hash graph heads.

6.3.7 BFT Strong Eventual Consistency

To achieve BFT strong eventual consistency, the CRDT algorithm must satisfy the following properties:

locale *bft-strong-eventual-consistency* = *peers-with-arbitrary-history* +
assumes *sem-check-only-ancestors-relevant*:
‹(*ancestor-nodes-of n*) ⊆ *fset G* ⟹ *is-sem-valid-set* (*ancestor-nodes-of n*) *n* ⟷ *is-sem-valid G n*›
assumes *concurrent-opers-commute*: ‹*hb.concurrent-ops-commute* (*delivered-nodes i*)›
assumes *step-never-fails*: ‹*apply-history* ([], {}) *ns* = (*dn*, *G*) ⟹ *no-failure dn* ⟹
check-and-apply (*dn*, *G*) (*hs*, *v*) = (*dn'*, *G'*) ⟹ *no-failure dn'*›

The *concurrent-ops-commute* property is required to ensure that the operations can be executed in any causal order without affecting the final state. The *only-ancestors-relevant* property is required to ensure that the validity of a node is consistent across different valid graphs, so that the Byzantine peers cannot prevent the correct peers from synchronizing their hash graph heads.

The convergence theorem and synchronization theorem prove that two correct peers can reach the same state, but this does not exclude the possibility that the state is a failure state. Therefore, the *step-never-fails* property, which is not discussed in the previous sections, is required to ensure that the correct peers never reach failure state. The *no-failure* is defined as follows:

definition *no-failure* :: ‹('*hash*, '*val*) *node list* ⇒ *bool*› **where**
‹*no-failure xs* = (*hb.apply-operations xs initial-state* ≠ *None*)›

By induction, we can prove that, for a correct peer, it never reaches a failure state:

theorem *sec-progress*: ‹*apply-operations* (*delivered-nodes i*) ≠ *None*›

By the lemma *convergence*, we can show that any two correct peers can reach the same state if they have the same heads.

theorem *sec-convergence*:
assumes ‹*heads* (*graph i*) = *heads* (*graph j*)›
shows ‹*apply-operations* (*delivered-nodes i*) = *apply-operations* (*delivered-nodes j*)›

By the lemma *synchronization*, we can show that any two correct peers can synchronize to the same heads.

theorem *sec-synchronization*:
assumes ‹*fset-of-list xs* = (*graph j*) − (*graph i*)›
and ‹*struct-valid-seq* (*graph i*) *xs*›
shows ‹*sem-valid-seq* (*graph i*) *xs*›

6.4 Common Vulnerabilities

We now give some common vulnerabilities for most CRDTs we discovered and approaches on how to address them.

6.4.1 Uniqueness of IDs

Many CRDTs require the uniqueness of IDs for operations, including ORSet, RGA, Logoot [WUM09], and WOOT [Ost+06]. In a non-Byzantine environment, this uniqueness is typically ensured by the combination of a local incrementing counter and a unique global peer ID. However, in a Byzantine environment, the uniqueness can be easily broken by assigning the same ID to different operations. We have shown how to exploit this vulnerability in Section 5.2.2.

Previsouly, we have proved the uniqueness of delivered nodes in lemma *apply-history-preserve-distinctness*. Since we have assumed that the hash function is collision-resistant, the hash of the node is also unique. Therefore, using the hash of the node as the ID of the operation contained in the node can ensure the uniqueness of IDs. If a hash node contains multiple operations, we can append the integers 1, 2,... to the hash to form a sequence of IDs for the operations in the node.

Some CRDTs, e.g., RGA, not only require the uniqueness of IDs for elements, but also require that IDs are ordered in a way that is consistent with causality (causally later operations have greater IDs). In Section 3.2.3, we show that *insert-body* skips over the elements that have greater IDs than the inserted element. RGA is designed to skip over those elements because greater IDs are considered to be causally after the inserted element and we should not insert the new element before them. However, hashes do not have this ordering property. In such case, we can use (i, h) pairs as IDs, ordered lexicographically, where i is an integer that is consistent with causality and h is the hash of the operation for uniqueness.

Another downside of using the hash of the node as the ID is that the size of the ID is typically 32 bytes, which is larger than other schemes. This can be mitigated by not storing the ID explicitly and only computing the ID on demand.

6.4.2 Dependency between Operations

Many CRDTs have operations that are dependent on other operations, and such operations must be applied after their dependent operations. For example, as discussed in Section 5.2.1, a Rem operation depends on the existence of the element to be removed. That is, the Add operation that adds the element to the set must be applied before the Rem operation.

To ensure the dependent operations are applied before the operations that depend on them, we can use the following heuristics:

- We do not allow operations that depend on concurrent operations. As discussed in Section 6.3.4, validity check must be deterministic and based solely on the operation itself and its ancestors in the hash graph.
- For an operation that depends on operations that are causally before it, and the the node containing the operation is structually valid with respect to the graph, if the dependent operations are not present in the history, we should discard the incoming operation. Since all the ancestor operations must have been applied, if the dependent operations are not present, those dependent operations will never be present in the future.

6.5 Conclusion

In this chapter, we have presented a comprehensive formalization of our system design, focusing on the use of hash graphs as the underlying data structure and proving sufficient conditions for achieving Byzantine Fault Tolerant (BFT) convergence for CRDTs.

With the formalization, we have shown that:

1. Any two correct peers can compare their heads and determine if their hash graphs are identical.
2. If the CRDT algorithm uses appropriate semantic checks to ensure the concurrent operations commute, two correct peers with identical heads must be in the same CRDT state.
3. If the semantic checks only depend on the ancestors of the incoming node, Byzantine peers cannot prevent correct peers from synchronizing their hash graph to have the same heads.

In conclusion, our formalization provides a solid foundation for Byzantine Fault Tolerant CRDTs design and verification. To make an existing CRDT algorithm Byzantine Fault Tolant, one needs to design an appropriate semantic check function, make necessary modifications to the definition and interpretation of the operations, and verify its correctness using our framework. In the following chapters, we will provide heuristics on how to design Byzantine Fault Tolant CRDTs, and show concrete implementations of Byzantine Fault Tolant CRDTs including ORSet and RGA, and prove their correctness using our framework.

7 Evaluation

To evaluate the effectiveness of our framework from Chapter 6 in designing and verifying Byzantine Fault Tolerant (BFT) CRDTs, we implemented two Byzantine Fault Tolerant CRDTs: the Observed-Remove Set (ORSet) and the Replicated Growable Array (RGA). Non-BFT forms of these algorithms are well-known, and were formally verified by Gomes et al. [Gom+17b]. We made minor modifications to these algorithms to ensure their convergence in a Byzantine environment, and prove their correctness using our framework.

7.1 Methodology

To design and verify Byzantine Fault Tolerant CRDTs, we follow an iterative process that combines analysis with formal verification. This process can be described as follows:

1. **Identify Potential Vulnerabilities**: Begin by analyzing the traditional CRDT algorithm to identify potential vulnerabilities that could be exploited by Byzantine peers. This step doesn't require an exhaustive list of the vulnerabilities.

2. **Propose Modifications**: Based on the identified vulnerabilities, propose modifications to the CRDT algorithm. These modifications should aim to preserve the core semantics of the CRDT while making it tolerant to attacks targeting on those vulnerabilities.

3. **Formalize the Modified Algorithm**: Formalize the modified CRDT algorithm using Isabelle/HOL. This involves defining the operations, state, and semantic validity checks.

4. **Attempt Verification**: Use the framework we developed in Chapter 6 to verify the correctness of the modified CRDT.

5. **Analyze Proof Failures**: If the verification attempt fails at any point, carefully analyze the proof state in Isabelle/HOL proof assistant to understand the reason for failure. This analysis often reveals subtle flaws in the modified algorithm.

6. **Refine the Algorithm**: Based on the insights gained from the failed proof attempt, refine the CRDT algorithm further. This might involve strengthening validity checks, modifying how operations are interpreted, or changing the structure of the state.

7. **Iterate**: Repeat steps 3-6 until a full proof of correctness is achieved. Each iteration brings the algorithm closer to being provably Byzantine fault tolerant.

By following this iterative process, we not only work towards creating robust BFT CRDT algorithms but also gain a deep, formally-grounded understanding of the vulnerabilities inherent in existing CRDT algorithms. This understanding could be potentially helpful for other BFT CRDT designs as some of the vulnerabilities are general and exist in most CRDTs.

7.2 Byzantine Fault Tolerant ORSet

Our definition is based on the non-BFT ORSet Isabelle definition of an operation-based ORSet shown in Section 3.2.2. Unlike the traditional ORSet, our BFT version does not include an element id in the *Add* operation. This modification is necessary because allowing the operation's creator to choose the ID would enable attackers to create multiple operations with the same value and ID but different predecessors, thus compromising the uniqueness of the ID.

The modified operations for our BFT ORSet are defined as follows:

datatype $('id, 'a)$ *operation* $=$ *Add* $\langle 'a \rangle$ | *Rem* $\langle 'id\ set \rangle$ $\langle 'a \rangle$

The state definition for our BFT ORSet remains unchanged from the traditional ORSet. The state is still represented as a map from elements to their corresponding identifiers:

type-synonym $('id, 'a)$ *state* $= \langle 'a \Rightarrow 'id\ set \rangle$

To facilitate our work with the ORSet operations stored in the hash graph nodes, we define several type synonyms:

type-synonym $('hash, 'a)$ *ORSetG* $= \langle ('hash, ('hash, 'a)\ operation)\ hash\text{-}graph \rangle$
type-synonym $('hash, 'a)$ *ORSetN* $= \langle ('hash, ('hash, 'a)\ operation)\ node \rangle$
type-synonym $('hash, 'a)$ *ORSetH* $= \langle ('hash, ('hash, 'a)\ operation)\ hash\text{-}func \rangle$
type-synonym $('hash, 'a)$ *ORSetC* $= \langle ('hash, ('hash, 'a)\ operation)\ causal\text{-}func \rangle$

The interpretation of operations in our BFT ORSet is defined as follows:

fun *interpret-op'* :: ‹('*hash*, '*a*) *ORSetH* ⇒ ('*hash*, '*a*) *ORSetN* ⇒ ('*hash*, '*a*) *state* ⇒ ('*hash*, '*a*) *state option*› **where**
‹*interpret-op' H* (*hs*, *oper*) *state* = (
let before = *state* (*op-elem oper*);
after = *case oper of*
Add e ⇒ *before* ∪ {*H* (*hs*, *oper*)}
| *Rem is e* ⇒ *before* − *is*
in Some (*state* ((*op-elem oper*) := *after*)))›

In this implementation, when interpreting an *Add* operation, we generate a unique ID for the element by hashing the node that contains the *Add* operation.

We also define the following predicate to check the semantic validity of a new node with respect to an existing state. An *Add* operation is always semantically valid and can always be added to the hash graph. A *Rem* operation is only valid if all the IDs it intends to remove are already present in its ancestors. This ensures that we only remove elements that have been previously added. Here *C* is a placeholder for a *ancestor* relation, which is not defined before instantiating the locale. *C n (hs, Rem is e)* means the *n* is an ancestor of the node that contains the *Rem* operation.

fun *is-orset-sem-valid* :: ‹('*hash*, '*a*) *ORSetC* ⇒ ('*hash*, '*a*) *ORSetH* ⇒ ('*hash*, '*a*) *ORSetN set* ⇒ ('*hash*, '*a*) *ORSetN* ⇒ *bool*› **where**
‹*is-orset-sem-valid C H S* (*hs*, *Add e*) = *True*›
‹*is-orset-sem-valid C H S* (*hs*, *Rem is e*) =
(∀ *i* ∈ *is*. ∃ *n* ∈ *S*. (*C n* (*hs*, *Rem is e*)) ∧ (*snd n* = *Add e*) ∧ (*H n* = *i*))›

To demonstrate that our modified ORSet can achieve consistency in a Byzantine environment, we instantiate the *peers-with-arbitrary-history* locale with the functions defined above, which allows us to prove the ORSet we defined converges under Byzantine conditions.

locale *bft-orset* = *peers-with-arbitrary-history H - interpret-op'* ‹λ*x*. {}› *is-orset-sem-valid* **for**
H :: ‹('*hash*, '*a*) *ORSetH*›

7.2.1 Concurrent Operations Commute

Add Operations Commute

For two concurrent *Add* operations, they commute with each other trivially, as the following lemma shows:

lemma *add-add-commute*:
shows ‹⟨(*hs1*, *Add e1*)⟩ ▷ ⟨(*hs2*, *Add e2*)⟩ = ⟨(*hs2*, *Add e2*)⟩ ▷ ⟨(*hs1*, *Add e1*)⟩›
by(*auto simp add*: *op-elem-def kleisli-def*, *fastforce*)

Add and *Rem* Operations commute

However, an *Add* operation and a *Rem* operation do not commute unconditionally. It holds only when the ID of the *Add* operation is not present in the set of IDs referenced by the *Rem* operation.

lemma *add-rem-commute*:
assumes ‹$H\ (hs1, Add\ e1) \notin is$›
shows ‹$\langle (hs1, Add\ e1) \rangle \rhd \langle (hs2, Rem\ is\ e2) \rangle = \langle (hs2, Rem\ is\ e2) \rangle \rhd \langle (hs1, Add\ e1) \rangle$›
using *assms* **by**(*auto simp add*: *kleisli-def op-elem-def*, *fastforce*)

To prove that a correct peer will never have a *Rem* operation that references an concurrent *Add* operation, we first prove that for any valid graph, when a *Rem* operation exists, the IDs it intends to remove must already exist in the graph.

lemma *rem-has-ancestor-ids*:
assumes ‹*valid-graph G*›
and ‹$x = (hs, Rem\ is\ e)$›
and ‹$x \in G$›
shows ‹$\forall i \in is.\ \exists n \in G.\ snd\ n = Add\ e \wedge H\ n = i$›

Proof. We prove this lemma by induction on the graph G.

For the base case, G is an empty graph, so the lemma holds trivially.

For the inductive step, if graph G' is obtained by extending graph G with a new node n, we consider two cases:

- If the new node contains a *Rem* operation, since semantic validity is checked, the new node is semantically valid with respect to the graph G. Therefore, all the IDs it intends to remove must already exist in the graph G, thus the lemma holds.
- If the new node contains an *Add* operation, by induction hypothesis the lemma holds.

□

We then prove that for any valid graph, if there is a node that contains an *Add* operation and a node that contains a *Rem* operation such that they are concurrent, the *Add* operation's ID will not appear in the set of IDs referenced by the *Rem* operation.

lemma *sem-valid-concurrent*:
assumes ‹*valid-graph G*›
and ‹$x = (hs1, (Add\ e1))$›
and ‹$y = (hs2, (Rem\ is\ e2))$›

and ‹$x \in G$›
and ‹$y \in G$›
and ‹*hb.concurrent x y*›
shows ‹$H\ (hs1, (Add\ e1)) \notin is$›

Proof. We prove this lemma by induction on the graph G.

For the base case, G is an empty graph, so the lemma holds trivially.

For the inductive step, if graph G' is obtained by extending graph G with a new node n, we consider two cases:

- If the new node contains an *Rem* operation, since semantic validity is checked, the new node is semantically valid with respect to the graph G. Therefore, all the IDs it intends to remove must already exist in the graph G, and those the *Add* operations that contain those IDs are ancestors of the new *Rem* operation. The hash graph doesn't contain two nodes with the same hash according to the no collision assumption, and the ID of an *Add* operation is the hash of the node that contains the *Add* operation. Therefore, it is impossible to have another node that is concurrent with the *Rem* operation and the ID that is referenced by the *Rem* operation.
- If the new node contains an *Add* operation, according to the structual validity check, the *Add* operation must not be in the old graph G before. According to lemma *rem-has-ancestor-ids*, the ids referenced by all the nodes that contain *Rem* operations in the old graph G are already present in the old graph G. Therefore, the new *Add* operation cannot be referenced by any *Rem* operation in the old graph G. By induction hypothesis, the lemma holds for the old graph G. Since the new *Add* operation is not referenced by any *Rem* operation in the old graph G, the lemma also holds for the new graph G'.

□

Rem and *Rem* Operations Commute

This lemma is trivially true, as shown below:

lemma *rem-rem-commute*:
shows ‹$\langle(hs1, Rem\ i1\ e1)\rangle \rhd \langle(hs2, Rem\ i2\ e2)\rangle = \langle(hs2, Rem\ i2\ e2)\rangle \rhd \langle(hs1, Rem\ i1\ e1)\rangle$›
by(*unfold op-elem-def kleisli-def*, *fastforce*)

By lemmas *add-add-commute*, *add-rem-commute*, *rem-rem-commute* and *sem-valid-concurrent*, we can derive that for any delivered nodes of a correct peer, if there are two concurrent operations, they must commute.

lemma *concurrent-operations-commute*:
assumes ‹*xs* = (*delivered-nodes i*)›
shows ‹*hb.concurrent-ops-commute xs*›

7.2.2 Synchronization

The semantic validity check depends solely on the new node's ancestors, as non-ancestor nodes are filtered out during this process.

lemma *sem-valid-only-ancestors-relevant*:
assumes ‹(*ancestor-nodes-of n*) ⊆ *fset G*›
shows ‹*is-sem-valid-set* (*ancestor-nodes-of n*) *n* ⟷ *is-sem-valid G n*›

7.2.3 No Failure

The *interpret-op'* function never fails, so the lemma holds trivially.

lemma *step-never-fails*:
assumes ‹*apply-history* ([], {}) *ns* = (*dn*, *G*)›
and ‹*no-failure dn*›
and ‹*check-and-apply* (*dn*, *G*) (*hs*, *v*) = (*dn'*, *G'*)›
shows ‹*no-failure dn'*›
using *assms* **proof** (*cases is-valid G* (*hs*, *v*))
case *True*
then show *?thesis* **using** *interpret-op'-never-fails apply-operations-def apply-operations-snoc*
assms(2) *assms*(3) *no-failure-def* **by** *auto*
next
case *False*
then show *?thesis*
by (*metis assms*(2) *assms*(3) *check-and-apply.simps fst-eqD*)
qed

7.2.4 BFT Strong Eventual Consistency

By lemmas *sem-valid-only-ancestors-relevant*, *step-never-fails* and *concurrent-operations-commute*, we can instantiate the *bft-strong-eventual-consistency* locale and prove that our BFT ORSet is correct under Byzantine environment.

sublocale *sec*: *bft-strong-eventual-consistency H - interpret-op'* ‹λ*x*. {}› *is-orset-sem-valid*
proof
fix *n G*
assume ‹*ancestor-nodes-of n* ⊆ *fset G*›
show ‹*is-sem-valid-set* (*ancestor-nodes-of n*) *n* = *is-sem-valid G n*›

```
  using ‹ancestor-nodes-of n ⊆ fset G› sem-valid-only-ancestors-relevant by presburger
next
 fix i
 show ‹hb.concurrent-ops-commute (delivered-nodes i)›
  by (simp add: concurrent-operations-commute)
next
 fix ns dn G hs v dn' G'
 show ‹apply-history ([], {}) ns = (dn, G) ⟹
    no-failure dn ⟹
    check-and-apply (dn, G) (hs, v) = (dn', G') ⟹ no-failure dn'›
  using step-never-fails by blast
qed
```

7.3 Byzantine Fault Tolerant RGA

We now adapt the RGA implementation from Section 3.2.3 to tolerate Byzantine faults. The ID of each element in RGA affects the position of the element in the list, since *insert-body* skips over the elements that have greater IDs than the inserted element. Therefore, we cannot simply use the hash of the node containing the Insert operation as its ID, as this would prevent the operation generator from controlling the exact position of the inserted element. Instead, we use a pair consisting of an ID that is chosen by the generating peer and the hash of the node containing the Insert operation as the element ID. When comparing element IDs, we first compare the IDs chosen by the peers. If they are identical, we then compare the hashes. This approach ensures that the operation generator can control the position of the inserted element, while still guaranteeing the uniqueness of the ID.

We define the operations for our BFT-RGA as follows:

```
type-synonym ('id, 'hash) elem-id = ‹('id × 'hash)›
derive linorder prod
datatype ('id, 'hash, 'v) operation =
 Insert ‹'v› ‹'id› ‹('id, 'hash) elem-id option› |
 Delete ‹('id, 'hash) elem-id›
```

Similar to the ORSet implementation, we define several type synonyms to work with RGA operations stored in hash graph nodes:

```
type-synonym ('id, 'hash, 'v) RGAG = ‹('hash, ('id, 'hash, 'v) operation) hash-graph›
type-synonym ('id, 'hash, 'v) RGAN = ‹('hash, ('id, 'hash, 'v) operation) node›
type-synonym ('id, 'hash, 'v) RGAH = ‹('hash, ('id, 'hash, 'v) operation) hash-func›
type-synonym ('id, 'hash, 'v) RGAC = ‹('hash, ('id, 'hash, 'v) operation) causal-func›
```

During the interpretation of the *Insert* operation, the ID is generated by hashing the node that contains the operation, and combining it with an ID chosen by the generating peer. We did not modify any definitions of the original Ordered-List algorithm described in Section 3.2.3. Instead, we reused it as is, preserving its core functionality while adapting it for our Byzantine fault-tolerant context.

```
fun interpret-op :: ‹('id::{linorder}, 'hash::{linorder}, 'v) RGAH ⇒ ('id, 'hash, 'v) RGAN
 ⇒ (('id, 'hash) elem-id, 'v) elt list ⇀ (('id, 'hash) elem-id, 'v) elt list› where
 ‹interpret-op H (hs, (Insert v i ei)) xs =
   (let h = H (hs, (Insert v i ei)) in
    insert xs ((i, h), v, True) ei
 )›
 ‹interpret-op H (hs, (Delete ei)) xs = delete xs ei›
```

We define a semantic validity check function for our BFT-RGA. This function ensures that:

- For Insert operations:
  - The preceding element's ID must exist in the current state.
  - The hash node that added the preceding element must be an ancestor of the new node.
  - The new node cannot reference itself as an ID (although this is computationally infeasible, we check for it explicitly).
- For Delete operations:
  - The ID to be deleted must exist in the current state.
  - The hash nodes that added the ID must be an ancestor of the new node.

The implementation of this semantic validity check is as follows:

```
fun is-rga-sem-valid :: ‹('id, 'hash, 'a) RGAC ⇒ ('id, 'hash, 'a) RGAH ⇒ ('id, 'hash, 'a) RGAN set
⇒
 ('id, 'hash, 'a) RGAN ⇒ bool› where
 ‹is-rga-sem-valid C H G (hs, Insert v i ei) = (
   case ei of
     None ⇒ True
   | Some ii ⇒ (∃hs' v' i' ei'.
       (hs', Insert v' i' ei') ∈ G ∧
       C (hs', Insert v' i' ei') (hs, Insert v i ei) ∧
       H (hs', Insert v' i' ei') = snd ii ∧
       i' = (fst ii)) ∧
```

```
      H (hs, Insert v i ei) ≠ snd ii
  )›
| ‹is-rga-sem-valid C H G (hs, Delete ei) = (∃hs' v' i' ei'.
    (hs', Insert v' i' ei') ∈ G ∧
    C (hs', Insert v' i' ei') (hs, Delete ei) ∧
    H (hs', Insert v' i' ei') = snd ei ∧
    i' = (fst ei)
)›
```

7.3.1 Concurrent Operations Commute

Insert Operations Commute

We reuse the lemmas proved by [Gom+17b] for the original Ordered-List algorithm. It shows that two concurrent *Insert* operations commute if they do not reference each other. For the detailed proof of this lemma, please refer to the source code of this work.

```
lemma insert-commutes:
  assumes ‹fst e1 ≠ fst e2›
        ‹i1 = None ∨ i1 ≠ Some (fst e2)›
        ‹i2 = None ∨ i2 ≠ Some (fst e1)›
  shows  ‹insert xs e1 i1 ⤜ (λys. insert ys e2 i2) =
         insert xs e2 i2 ⤜ (λys. insert ys e1 i1)›
```

To ensure the commutativity of concurrent *Insert* operations in our BFT-RGA, we need to prove that for any correct peer, there will never be two concurrent *Insert* operations that reference each other.

We first prove that for any valid graph, the ID referenced by an *Insert* operation must already exist in the graph. This is formalized in the following lemma:

```
lemma insert-has-pred-id:
  assumes ‹valid-graph G›
    and ‹x ∈ G›
    and ‹x = (hs, Insert v i (Some ei))›
  shows ‹∃y ∈ G. y ≺ x ∧ H y = snd ei›
```

Proof. We prove this lemma by induction on the graph G.

For the base case, G is an empty graph, so the lemma holds trivially.

For the inductive step, if graph G' is obtained by extending graph G with a new node n, since the new node is semantically valid with respect to the graph G, the ID the new node references must already exist in the graph G. Therefore, the lemma holds.

□

Next, we prove that for any two concurrent *Insert* operations, they do not reference each other. This is formalized in the following lemma:

lemma *insert-commute-id*:
assumes ‹*valid-graph G*›
and ‹$x = (hs1, Insert\ v1\ i1\ ei1)$›
and ‹$y = (hs2, Insert\ v2\ i2\ ei2)$›
and ‹$x \in G$›
and ‹$y \in G$›
and ‹*hb.concurrent x y*›
shows ‹$ei2 = None \vee ei2 \neq Some\ (h, H\ x)$›

Proof. We prove this lemma by induction on the graph G.

For the base case, G is an empty graph, so the lemma holds trivially.

For the inductive step, if graph G' is obtained by extending graph G with a new node n, we consider two cases:

- If the new node contains a *Delete* operation, by the induction hypothesis, the lemma holds for the old graph G. Since the new node does not contain an *Insert* operation, the lemma holds for the new graph G'.
- If the new node contains an *Insert* operation, by the induction hypothesis, the lemma holds for the old graph G. Since the new *Insert* operation passes the semantic validity check, the ID it references must be in old graph G and the hash node containing the referenced *Insert* operation must be an ancestor of the new node, thus they are not concurrent. Furthermore, according to lemma *insert-has-pred-id*, the operations in the old graph G do not reference the new node, therefore the lemma holds for the new graph G'.

□

By lemmas *insert-commutes* and *insert-commute-id*, we can deduce that for any valid graph, if there are two concurrent *Insert* operations, they must commute.

lemma *ins-ins-commute*:
assumes *valid-graph G*
and ‹$x = (hs1, Insert\ v1\ i1\ ei1)$›
and ‹$y = (hs2, Insert\ v2\ i2\ ei2)$›
and ‹$x \in G$›
and ‹$y \in G$›
and ‹*hb.concurrent x y*›
shows ‹$\langle x \rangle \rhd \langle y \rangle = \langle y \rangle \rhd \langle x \rangle$›

Insert and *Delete* Operations Commute

Similar to the previous case, we can reuse a lemma proved in the Ordered-List theory which demonstrates that if a delete operation does not reference the ID of an insert operation, then these operations commute.

lemma *insert-delete-commute*:
assumes ‹*i2* ≠ *fst e*›
shows ‹*insert xs e i1* ⤜ (λ*ys*. *delete ys i2*) = *delete xs i2* ⤜ (λ*ys*. *insert ys e i1*)›

To prove that concurrent *Insert* and *Delete* operations commute, we first need to prove that for any valid graph G, when a *Delete* operation exists in G, the ID of the *Insert* operation it references also exists in G. We formalize this as a lemma:

lemma *delete-has-pred-id*:
assumes ‹*valid-graph G*›
and ‹*x* ∈ *G*›
and ‹*x* = (*hs*, *Delete ei*)›
shows ‹∃ *y* ∈ *G*. *y* ≺ *x* ∧ *H y* = *snd ei*›

Proof. We prove this lemma by induction on the graph G.

For the base case, G is an empty graph, so the lemma holds trivially.

For the inductive step, let graph G' be obtained by extending graph G with a new node n. Since the new node is semantically valid with respect to the graph G, the ID new node references must already exist in the graph G. Therefore, the lemma holds. □

We then prove that for any valid graph, if there is a node that contains an *Insert* operation and a node that contains a *Delete* operation that are concurrent, the *Insert* operation's ID will not appear as the ID referenced by the *Delete* operation.

lemma *ins-del-commute-id*:
assumes ‹*valid-graph G*›
and ‹*x* = (*hs1*, *Insert v1 i1 ei1*)›
and ‹*y* = (*hs2*, *Delete ei2*)›
and ‹*x* ∈ *G*›
and ‹*y* ∈ *G*›
and ‹*hb.concurrent x y*›
shows ‹*snd ei2* ≠ *H x*›

Proof. We prove this lemma by induction on the graph G. For the base case, G is an empty graph, so the lemma holds trivially. For the inductive step, if graph G' is obtained by extending graph G with a new node n, we consider two cases:

- If the new node contains a *Delete* operation, by induction hypothesis, the lemma holds for the old graph *G*. Since the new node references an ID that exists in old graph *G*, and the node that hashes to that ID is an ancestor of the new node, the new node cannot reference a concurrent *Insert* operation.
- If the new node contains an *Insert* operation, according to lemma *delete-has-pred-id*, the nodes that contain *Delete* operations in the old graph *G* reference IDs that already exist in the old graph *G*. Therefore, the new node cannot be referenced by any *Delete* operation in the old graph *G'*, thus the lemma holds.

□

From lemmas *insert-delete-commute* and *ins-del-commute-id*, we can derive that for any valid graph, if there are concurrent *Insert* and *Delete* operations, they must commute.

lemma *ins-del-commute*:
assumes *valid-graph G*
and ‹$x = (hs1, Insert\ v1\ i1\ ei1)$›
and ‹$y = (hs2, Delete\ ei2)$›
and ‹$x \in G$›
and ‹$y \in G$›
and ‹*hb.concurrent x y*›
shows ‹$\langle x \rangle \rhd \langle y \rangle = \langle y \rangle \rhd \langle x \rangle$›

Delete Operations Commute

Similarly, we can reuse a lemma proved in the Ordered-List theory which demonstrates that if two *Delete* operations commute under no conditions.

lemma *delete-commutes*:
shows ‹$delete\ xs\ i1 \gg\!= (\lambda ys.\ delete\ ys\ i2) = delete\ xs\ i2 \gg\!= (\lambda ys.\ delete\ ys\ i1)$›

Therefore, we can easily derive that in our *BFT-RGA*, two *Delete* operations always commute.

lemma *del-del-commute*:
shows ‹$\langle (hs1, Delete\ ei1) \rangle \rhd \langle (hs2, Delete\ ei2) \rangle = \langle (hs2, Delete\ ei2) \rangle \rhd \langle (hs1, Delete\ ei1) \rangle$›

From lemmas *ins-ins-commute*, *ins-del-commute*, and *del-del-commute*, we can derive that for any delivered nodes of a correct peer, two concurrent operations must commute.

lemma *concurrent-operations-commute*:
assumes ‹$xs = (delivered\text{-}nodes\ i)$›
shows ‹*hb.concurrent-ops-commute xs*›

7.3.2 Synchronization

The semantic validity check depends solely on the new node's ancestors, as non-ancestor nodes are filtered out during this process.

lemma *sem-valid-only-ancestors-relevant*:
assumes ‹(*ancestor-nodes-of n*) ⊆ *fset G*›
shows ‹*is-sem-valid-set* (*ancestor-nodes-of n*) *n* ⟷ *is-sem-valid G n*›

7.3.3 No Failure

We first define the function *inserted-ids* to get all the IDs of the *Insert* operations in the delivered nodes.

definition *inserted-ids* :: ‹('*id*, '*hash*, '*v*) *RGAN list* ⇒ ('*id*, '*hash*) *elem-id list*› **where**
‹*inserted-ids xs* ≡
(*List.map-filter* (λ*x*. *case x of*
(-, *Insert v i ei*) ⇒ *Some* (*i*, *H x*)
- ⇒ *None*))
xs›

We then define the function *element-ids* to get all the IDs of the elements in the CRDT state.

definition *element-ids* :: ‹(('*id* × '*hash*) × '*v* × *bool*) *list* ⇒ ('*id* × '*hash*) *set*› **where**
‹*element-ids list* ≡ *set* (*map fst list*)›

We can prove that for any correct peer, if the *inserted-ids* of the *Insert* operations in the delivered nodes are the same as the IDs of the elements in the CRDT state before applying any history, then they are still the same after applying any history.

lemma *apply-opers-idx-elems*:
assumes ‹*apply-history* (*dn*, *G*) *ns* = (*dn'*, *G'*)›
and ‹*apply-operations dn* = *Some es*›
and ‹*element-ids es* = *set* (*inserted-ids dn*)›
and ‹*apply-operations dn'* = *Some es'*›
shows ‹*element-ids es'* = *set* (*inserted-ids dn'*)›

We then prove that for any correct peer, if the *inserted-ids* of the *Insert* operations in the delivered nodes are the same as the IDs of the elements in the CRDT state before applying any operation, then the state must not be failure after applying any operation.

lemma *step-never-fails-with-idx-elems*:
assumes ‹*check-and-apply* (*dn*, *G*) (*hs*, *v*) = (*dn'*, *G'*)›
and ‹*fset-of-list dn* = *G*›
and ‹*apply-operations dn* = *Some es*›
and ‹*element-ids es* = *set* (*inserted-ids dn*)›
shows ‹*no-failure dn'*›

By the above lemmas *apply-opers-idx-elems* and *step-never-fails-with-idx-elems*, we can prove that a correct peer never fails to check and apply any operation.

lemma *step-never-fails*:
assumes ‹*apply-history* ([], {}) *ns* = (*dn*, *G*)›
and ‹*no-failure dn*›
and ‹*check-and-apply* (*dn*, *G*) (*hs*, *v*) = (*dn'*, *G'*)›
shows ‹*no-failure dn'*›

7.3.4 BFT Strong Eventual Consistency

By lemmas *sem-valid-only-ancestors-relevant*, *step-never-fails* and *concurrent-operations-commute*, we can instantiate the *bft-strong-eventual-consistency* locale and prove that our BFT-RGA is correct under Byzantine environment.

sublocale *sec*: *bft-strong-eventual-consistency H - interpret-op* ‹[]› *is-rga-sem-valid*
proof
fix *n G*
assume ‹*ancestor-nodes-of n* ⊆ *fset G*›
show ‹*is-sem-valid-set* (*ancestor-nodes-of n*) *n* = *is-sem-valid G n*›
using ‹*ancestor-nodes-of n* ⊆ *fset G*› *sem-valid-only-ancestors-relevant* **by** *presburger*
next
fix *i*
show ‹*hb.concurrent-ops-commute* (*delivered-nodes i*)›
by (*simp add*: *concurrent-operations-commute*)
next
fix *ns dn G hs v dn' G'*
show ‹*apply-history* ([], {}) *ns* = (*dn*, *G*) ⟹
no-failure dn ⟹
check-and-apply (*dn*, *G*) (*hs*, *v*) = (*dn'*, *G'*) ⟹ *no-failure dn'*›
using *step-never-fails* **by** *blast*
qed

7.4 Conclusion

In this chapter, we have presented the design and verification of Byzantine Fault Tolerant (BFT) versions of the RGA and ORSet CRDTs within our framework. We show

how to modify the original algorithms and prove the convergence and synchronization properties of these BFT CRDTs.

The only two undischarged assumptions in our formalization are that there are no collisions of the hash function, and that the attackers cannot construct two hash nodes that are ancestors of each other (i.e. introduce cycles into the hash DAG). Both assumptions can be enforced by using a cryptographic hash function like SHA-256 or SHA-3 in practice.

We used 1395 lines of code to formalize the framework, 244 lines of code for the BFT-ORSet and 522 lines of code for the BFT-RGA. We spent about 20 to 60 hours each on the analysis and formalization of the BFT-ORSet and BFT-RGA. The process involves understanding the original algorithms' vulnerabilities, modifying them to achieve the BFT property, and proving the convergence and synchronization properties of these BFT CRDTs. It took about 2 to 4 iterations to fix the vulnerabilities and semantic issues. On our laptop with 8 Core CPU and 32 GB RAM, it takes 10 seconds to verify the proof in Isabelle/HOL.

8 Conclusion

8.1 Summary

This thesis has presented a formal framework for designing and verifying BFT CRDTs. We developed a methodology for adapting existing CRDT algorithms to acheive SEC in the presence of Byzantine faults, and we have provided formal proofs of the framework in Isabelle/HOL.

Our work builds upon the foundation laid by Gomes et al. [Gom+17b], who formalized the correctness of non-Byzantine CRDTs. We extended this approach to handle Byzantine environments, where some peers may behave arbitrarily. The key contributions of this thesis are:

- A formal model of a Byzantine system, which allows us to reason about the convergence of correct peers in the presence of Byzantine faults.
- A proof framework for verifying the correctness of BFT CRDT algorithms, including both convergence and the ability to synchronize in the presence of Byzantine faults.
- A methodology for adapting existing CRDT algorithms to acheive Byzantine Fault Tolerance.
- Concrete examples of how to apply this framework to create Byzantine fault-tolerant versions of two well-known CRDTs: the Observed-Remove Set (ORSet) and the Replicated Growable Array (RGA).

The BFT versions of ORSet and RGA that we developed demonstrate the applicability of our framework. By making relatively minor modifications to the original algorithms and proving their correctness within our framework, we were able to create CRDTs that tolerate Byzantine faults while preserving their original semantics.

Our work firstly provides a formal foundation for the study of BFT CRDTs, and the proofs can be used to verify the correctness of future CRDT algorithms.

8.2 Future Work

This thesis is only focused on the Strong Eventual Consistency (SEC) of correct peers in Byzantine environment. In the following, we will discuss some possible extensions to this work.

A BFT Access Control List (ACL) CRDT can be achieved by using the framework we proposed. With ACL CRDT support, the peers are able to limit the participants of the peer-to-peer system in a decentralized manner. Otherwise, a malicious peer may arbitrarily modify the content of the state.

The size of hash DAG increases as the number of operations increases. In the future, we can consider compacting the hash DAG to reduce the storage cost. A possible approach is to retain only heads of the entire hash DAG, and the corresponding CRDT state, which needs to be agreed by all members in the group enforced by ACL.

In this thesis we only provide several common vulnerabilities of most CRDTs. Although they are enough to guide the design of BFT ORSet and RGA, there might be more attacks to other CRDTs. In the future, we can consider more sophisticated CRDTs to find out more common vulnerabilities.

In the future we plan to export the Isabelle/HOL definitions to executable code. Currently, we have not done this because the *no-cycle* assumption stops Isabelle from exporting the code. Although the exported code usually has low performance, it can be used as a test reference for the correctness of performant implementations. That is, one can implement the BFT CRDTs according to the formalization and verify their correctness using differential testing.

Besides, the performance impact of retrofitting BFT to existing CRDTs is not considered in this thesis. We conjecture that the performance bottleneck lies in the ancestor traversal of the hash DAG. Many functions we discussed earlier uses ancestor check. A future project could focus on optimizing the performance of BFT CRDTs.

List of Figures

5.1 Byzantine peer r sends different updates A and B to p and q, using the same ID (r, 1). This causes p and q to falsely believe they have identical update sets, despite having different $\mathcal{U}_p$ and $\mathcal{U}_q$. 20
5.2 Byzantine peer r sends two concurrent updates to p and q in different order. The Add operation is applied to p, followed by the Remove operation, resulting in an empty set. However, q applies the Remove operation first, which is invalid and ignored by p. Followed by the Add operation, the state becomes {"Hello" : $[id_1]$}, leading to divergence. . . 22
5.3 Byzantine peer r sends two Insert operations to p and q with the same ID but different elements A and B. p and q apply the operation from r first, resulting in [A] and [B] respectively. Then, p and q exchange their updates. Since the ID is already present in the history, both p and q will consider the update as invalid and ignore it, leading to divergence. . . . 23
5.4 Byzantine peer r sends two concurrent Insert operations to p and q where one operation references the ID of the other operation. p applies the Insert A operation first, followed by the Insert B operation, resulting in [A, B]. q applies the Insert B operation first, which is an invalid operation as it references an ID that has not been added to the history yet, resulting in []. q then applies the Insert A operation, resulting in [A]. As a result, p and q diverge. 24

Bibliography

[And+13] L. André, S. Martin, G. Oster, and C.-L. Ignat. "Supporting adaptable granularity of changes for massive-scale collaborative editing." In: *9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*. IEEE. 2013, pp. 50–59.

[App11] AppJet, Inc. *Etherpad and EasySync Technical Manual*. Mar. 2011.

[Aut] Automerge Community. *Automerge CRDT*. `https://automerge.org`. Accessed: 2023-11-07.

[BAL16] C. Baquero, P. S. Almeida, and C. 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*. 2016, pp. 1–3.

[Ban+19] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis. "SoK: Consensus in the age of blockchains." In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 2019, pp. 183–198.

[Bie+12] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. "Brief announcement: Semantics of eventually consistent replicated sets." In: *International Symposium on Distributed Computing*. Springer. 2012, pp. 441–442.

[CL+99] M. Castro, B. Liskov, et al. "Practical Byzantine fault tolerance." In: *OsDI*. Vol. 99. 1999. 1999, pp. 173–186.

[Day10] J. Day-Richter. *What's different about the new Google Docs: Making collaboration fast*. Sept. 2010.

[DK24] L. Da and M. Kleppmann. *A Framework for Designing and Verifying Byzantine Fault Tolerant CRDTs*. Version 0.0.1. `https://github.com/LiangrunDa/bft-crdt-isabelle`. Oct. 2024.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. "Consensus in the presence of partial synchrony." In: *Journal of the ACM (JACM)* 35.2 (1988), pp. 288–323.

[Dou02] J. R. Douceur. "The Sybil attack." In: *International workshop on peer-to-peer systems*. Springer. 2002, pp. 251–260.

[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*. 1989, pp. 399–407.

[GK24] J. Gentle and M. Kleppmann. "Collaborative Text Editing with Eg-walker: Better, Faster, Smaller." In: *arXiv preprint arXiv:2409.14252* (2024).

[Gom+17a] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. "A framework for establishing Strong Eventual Consistency for Conflict-free Replicated Datatypes." In: *Archive of Formal Proofs* (July 2017). `https://isa-afp.org/entries/CRDT.html`, Formal proof development. ISSN: 2150-914x.

[Gom+17b] V. B. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. "Verifying strong eventual consistency in distributed systems." In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–28.

[Imi+03] A. Imine, P. Molli, G. Oster, and M. Rusinowitch. "Proving correctness of transformation functions in real-time groupware." In: *ECSCW 2003: Proceedings of the Eighth European Conference on Computer Supported Cooperative Work 14–18 September 2003, Helsinki, Finland*. Springer. 2003, pp. 277–293.

[Imi+06] A. Imine, M. Rusinowitch, G. Oster, and P. Molli. "Formal design and verification of operational transformation algorithms for copies convergence." In: *Theoretical Computer Science* 351.2 (2006), pp. 167–183.

[KB17] M. Kleppmann and A. R. Beresford. "A conflict-free replicated JSON datatype." In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (2017), pp. 2733–2746.

[KH20] M. Kleppmann and H. Howard. "Byzantine eventual consistency and the fundamental limits of peer-to-peer databases." In: *arXiv preprint arXiv:2012.00472* (2020).

[Kle+14] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. "Comprehensive formal verification of an OS microkernel." In: *ACM Transactions on Computer Systems (TOCS)* 32.1 (2014), pp. 1–70.

[Kle22] M. Kleppmann. "Making CRDTs Byzantine fault tolerant." In: *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*. 2022, pp. 8–15.

[Kue+23] C. Kuessner, R. Mogk, A.-K. Wickert, and M. Mezini. "Algebraic Replicated Data Types: Programming Secure Local-First Software." In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2023.

[Lam78] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565. DOI: 10.1145/359545.359563.

[Lit+22] G. Litt, S. Lim, M. Kleppmann, and P. Van Hardenberg. "Peritext: A CRDT for collaborative rich text editing." In: *Proceedings of the ACM on Human-Computer Interaction* 6.CSCW2 (2022), pp. 1–36.

[LL04] D. Li and R. Li. "Preserving operation effects relation in group editors." In: *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*. 2004, pp. 457–466.

[LSP19] L. Lamport, R. Shostak, and M. Pease. "The Byzantine generals problem." In: *Concurrency: the works of Leslie Lamport*. 2019, pp. 203–226.

[Lv+16] X. Lv, F. He, W. Cai, and Y. Cheng. "An efficient collaborative editing algorithm supporting string-based operations." In: *2016 IEEE 20th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE. 2016, pp. 45–50.

[Mey23] A. Meyer. "Range-Based Set Reconciliation." In: *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2023, pp. 59–69.

[MUW10] S. Martin, P. Urso, and S. Weiss. "Scalable XML Collaborative Editing with Undo: (Short Paper)." In: *On the Move to Meaningful Internet Systems: OTM 2010: Confederated International Conferences: CoopIS, IS, DOA and ODBASE, Hersonissos, Crete, Greece, October 25-29, 2010, Proceedings, Part I*. Springer. 2010, pp. 507–514.

[MV15] C. Meiklejohn and P. Van Roy. "Lasp: A language for distributed, coordination-free programming." In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. 2015, pp. 184–195.

[Nak08] S. Nakamoto. "Bitcoin: A peer-to-peer electronic cash system." In: *Decentralized business review* (2008).

[Néd+13] B. Nédelec, P. Molli, A. Mostefaoui, and E. Desmontils. "LSEQ: an adaptive structure for sequences in distributed collaborative editing." In: *Proceedings of the 2013 ACM Symposium on Document Engineering*. 2013, pp. 37–46.

[Nic+16] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma. "Near real-time peer-to-peer shared editing on extensible data types." In: *Proceedings of the 2016 ACM International Conference on Supporting Group Work*. 2016, pp. 39–49.

[Nic+95] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. "High-latency, low-bandwidth windowing in the Jupiter collaboration system." In: *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*. 1995, pp. 111–120.

[Nip13] T. Nipkow. "Programming and proving in Isabelle/HOL." In: *Technical report, University of Cambridge*. 2013.

[NMM16] B. Nédelec, P. Molli, and A. Mostefaoui. "Crate: Writing stories together with our browsers." In: *Proceedings of the 25th International Conference Companion on World Wide Web*. 2016, pp. 231–234.

[Ost+05] G. Oster, P. Urso, P. Molli, and A. Imine. *Proving correctness of transformation functions in collaborative editing systems*. Tech. rep. INRIA, 2005.

[Ost+06] G. Oster, P. Urso, P. Molli, and A. Imine. "Data consistency for P2P collaborative editing." In: *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. 2006, pp. 259–268.

[Par+83] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. "Detection of mutual inconsistency in distributed systems." In: *IEEE transactions on Software Engineering* 3 (1983), pp. 240–247.

[Pre+09] N. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia. "A commutative replicated data type for cooperative editing." In: *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE. 2009, pp. 395–403.

[RNG96] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. "An integrating, transformation-oriented approach to concurrency control and undo in group editors." In: *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*. 1996, pp. 288–297.

[Roh+11] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. "Replicated abstract data types: Building blocks for collaborative applications." In: *Journal of Parallel and Distributed Computing* 71.3 (2011), pp. 354–368.

[SCF98] M. Suleiman, M. Cart, and J. Ferrié. "Concurrent operations in a distributed and mobile collaborative environment." In: *Proceedings 14th International Conference on Data Engineering*. IEEE. 1998, pp. 36–45.

[SG21] E. Summermatter and C. Grothoff. "Byzantine Fault Tolerant Set Reconciliation." In: *Structure* 6 (2021), p. 1.

[Sha+11a] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of convergent and commutative replicated data types*. Tech. rep. Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[Sha+11b] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-free replicated data types." In: *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer. 2011, pp. 386–400.

[Wan+15] D. Wang, A. Mah, S. Lassen, and S. Thorogood. *Apache Wave (incubating) Protocol Documentation, Release 0.4*. Apache Software Foundation. Aug. 2015.

[WGK23] M. Weidner, J. Gentle, and M. Kleppmann. "The Art of the Fugue: Minimizing Interleaving in Collaborative Text Editing." In: *arXiv preprint arXiv:2305.00583* (2023).

[Wui+24] P. Wuille et al. *Minisketch: a library for BCH-based set reconciliation*. Version 0.0.1. `https://github.com/sipa/minisketch`. 2024.

[WUM07] S. Weiss, P. Urso, and P. Molli. "Wooki: a p2p wiki-based collaborative writing tool." In: *International Conference on Web Information Systems Engineering*. Springer. 2007, pp. 503–512.

[WUM09] S. Weiss, P. Urso, and P. Molli. "Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks." In: *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE. 2009, pp. 404–412.

[YGA24] L. Yang, Y. Gilad, and M. Alizadeh. "Practical rateless set reconciliation." In: *Proceedings of the ACM SIGCOMM 2024 Conference*. 2024, pp. 595–612.