HAW HAMBURG

# Bachelorarbeit

Micha Rosenbaum

## Java CRDT Framework for Autonomous Robots

*Fakultät Technik und Informatik*
*Department Informatik*

*Faculty of Computer Science and Engineering*
*Department Computer Science*

Micha Rosenbaum

# Java CRDT Framework for Autonomous Robots

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Axel Schmolitzky

Eingereicht am: August 19, 2021

**Micha Rosenbaum**

**Thema der Arbeit**

Java CRDT Framework for Autonomous Robots

**Stichworte**

CRDT, Gossip, Framework

**Kurzzusammenfassung**

Verteilte Kommunikation in kleinen Gruppen autonomer Roboter haben Probleme mit der Konsistenz und Verfügbarkeit bei Netzwerk-Partitionen (CAP Theorem). Entwickler:innen sollten diese Herausforderung nicht in allen Aspekten bewältigen müssen. Conflict-free Replicated Data Types (CRDTs) reduzieren den Effekt des CAP Theorems. In dieser Arbeit wird eine objektorientierte CRDT Framework-Architektur basierend auf dem Selective Hearing Modell designt. Das Framework dient als Middleware-Schicht, die von Entwickler:innen transparent genutzt werden kann.

**Micha Rosenbaum**

**Title of Thesis**

Java CRDT Framework for Autonomous Robots

**Keywords**

CRDT, Gossip, Framework

**Abstract**

Distributed communication in a small group of autonomous robots faces consistency and availability issues in the presence of network partitions (CAP theorem). Application developers should not be required to deal with all aspects of this challenge. Conflict-free replicated data types (CRDTs) reduce the effect of the CAP theorem. This thesis designs an object-oriented CRDT framework architecture based on the Selective Hearing programming model. This framework serves as a middleware layer that can be used by application developers transparently.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Acronyms

**API** application programming interface. 21, 22

**C-CRDT** Computational CRDT. 4, 10, 11, 17

**CaDS** Communication and Distributed Systems. 1, 2, 4, 44, 46, 57, *Glossary:* CaDS group

**CRDT** Conflict-free Replicated Data Type. 1–4, 6–11, 17–22, 24–26, 29–42, 45, 60–62

**CSV** comma-separated values. 46

**DRY [54, 58]** do not repeat yourself. 25, *Glossary:* DRY principle

**FST** fast-serialization. 39, 42

**HAW** Hochschule für Angewandte Wissenschaften. 4, *Glossary:* HAW Hamburg

**HyParView** Hybrid Partial View. 6, 12, 16, 19, 20, 22, 28, 39–42, 46, 49, 54, 56–62, *Glossary:* HyParView

**JDK** Java Development Kit. 39

**JSON** JavaScript Object Notation. 18, 39, 41, 62

**LDH** Last Delivery Hop. 4, 5, 47, 55, 56, 58, 60, 61

**Loomo** Segway Loomo Robot. 1, 2, 4, 18, 44, 58, *Glossary:* Loomo

**LWW** Last-Write-Wins. 10

**OOP** object-oriented programming. 31, 33, 35, 37

**Plumtree** Push-lazy push multicast tree. 2, 6, 12, 17, 19–22, 28, 29, 39–41, 45, 46, 49, 50, 53, 55–61

**RMI** Remote Method Invocation. 20

**RMR** Relative Message Redundancy. 4, 5, 47, 52–55, 58, 61

**RPC** Remote Procedure Call. 20

**SDK** Software Development Kit. 1

**TCP** Transmission Control Protocol. 16, 26, 42

**TDD** test-driven development. 62

**TTL** time-to-live. 16, 46, 57, 61

**UUID** universally unique identifier. 39, 40

# 1 Introduction

The field of autonomous robots is an important, evolving area that may shape the future. Because they often need to communicate with other robots or systems to perform their behavior or task, autonomous robots are often also distributed systems. The robot groups might not be known in advance. Instead, each robot maintains an ad-hoc neighbor list. The robots form an unstructured system for peer-to-peer communication within the group. In the last decade, many algorithms and communication protocols have been introduced, which are very valuable in unstructured, decentralized systems. Conflict-free Replicated Data Types (CRDTs) are promising technologies for replicating data in unstructured systems which communicate using gossip protocols.

Some of the Communication and Distributed Systems (CaDS) group[1] projects at the HAW Hamburg involve a fleet of Segway Loomo Robots (Loomos) [46]. The Loomo, as shown in Figure A.1, is an advanced robot that can be used as a smart personal vehicle. It includes many practical functions powered by artificial intelligence. Additionally, an Android Software Development Kit (SDK) is available, which can be used to implement custom Loomo applications [17]. Since Loomos are mobile by design, the CaDS group is using its small fleet as a case study to implement and research aspects of unstructured distributed systems.

## 1.1 Problem Statement

In distributed systems, concurrent write requests to data can occur on different nodes. It needs to be made certain that these concurrent updates do not produce conflicts in the data. This can be ensured in structured systems by applying a centralized setup that brings the updates into a global order. A central coordinator can permit an update to a single node at a time. In a primary-based setup, the primary node can perform an update and write data, while other nodes only have read-access to data. Unstructured, decentralized systems have to find other ways to coordinate their updates. A common way is to create ad-hoc coordination using consensus protocols with leader-election (such as Raft [42] or Paxos [22, 23]). Coordinated approaches have consistency or availability limitations when network partitions occur (CAP theorem [15]). This is an issue because a distributed application running on multiple nodes in an unstructured system should stay available on each node, regardless of the other nodes' statuses. Each node should be

[1] https://cads.informatik.haw-hamburg.de/, accessed May 2021

able to continue providing its service to the user (as 'local-first' software [21]). When two nodes recover from a temporary network partition and communicate with each other again, both should automatically and consistently reach the same state, even if they have received independent updates. These demands have been formalized as strong eventual consistency [48].

The example that is motivating this thesis, is the CaDS Loomo fleet. In this context, an application that measures the distance the entire fleet has traveled could be conceptualized. Each user would expect an individual Loomo to contribute its distance, no matter the network conditions. An active Loomo should not be affected if another Loomo from the network is no longer reachable or is turned off. For example, it would be unacceptable for Alice to not see the updated distance on her Loomo simply because Bob has turned his Loomo off. In the same sense, the application on Alice's Loomo should continue to function correctly, even if she drives in an area without network access. When the connection with the fleet is reestablished, the Loomos should calculate the same distance without losing any data. Currently, there is no system performing coordination-free distributed computations with all the Loomos in the CaDS group.

A CRDT allows a distributed system to achieve strong eventual consistency [48]. CRDTs provide "consistency without consensus."[4] Efficient and reliable data dissemination can be achieved by hybrid gossip protocols [36]. Push-lazy push multicast tree (Plumtree) [27] is such a hybrid gossip protocol. It combines fast, tree-based broadcasts with the more reliable gossip dissemination pattern [28].

The Selective Hearing programming model, introduced in [35], combines a programming layer based on CRDTs with a communication layer based on Plumtree. Selective Hearing has been specified in the functional programming paradigm, and implemented as part of the Lasp language in Erlang.[2] As with Plumtree, Selective Hearing has been evaluated using thousands of nodes. The named components are discussed in more detail in the next chapter.

This thesis describes the design and implementation of a proof-of-concept Java CRDT framework. With this proof-of-concept, the reliability of Selective Hearing in a context with only a few nodes, is investigated. The CaDS group's Loomo fleet contains three Loomos, whereas a typical Segway group tour through Hamburg usually includes 12–20 Segways. In order to use them on a Loomo, the Selective Hearing algorithms, CRDT implementations, and the used gossip protocols need to get ported to Java.

## 1.2 Goals and Requirements

According to [53, section 1.2, page 7], the goals of distributed systems can be defined as "supporting resource sharing," "making distribution transparent," "being open," and "being

[2] https://github.com/lasp-lang/lasp, accessed 23rd April 2021

scalable." This framework implements a middleware layer [53], to serve a developer who is building distributed applications. The middleware provides an interface to the developer, which hides the complexities of distributed systems—thus making distribution transparent. This framework's interface is the supplied CRDT library.

The following list introduces the assumed requirements from a fictional application developer's viewpoint, that should be addressed in this thesis.

1. Creating CRDT instances.
2. Merging CRDT replicas.
3. Merging only replicas with the same origin CRDTs.
4. Installing applications locally and in a distributed system, without requiring code changes.
5. Transparent synchronization of the CRDT state.

These requirements allow the distribution transparencies, that the framework needs to provide, to be derived. Requirements 1 and 2 call for the implementation of usable CRDTs. The first two basic requirements together with requirement 4 suggest the need of *access* transparency. An application using the CRDT interfaces should not need to differentiate between local or distributed CRDTs. To achieve requirement 5, the *replication* and *concurrency* transparencies need to be carried out. The combination of requirements 1, 4, and 5 suggest the implementation of a creational design pattern. The options discussed in this thesis are the factory method, abstract factory, and builder design patterns [13, 50]. Requirement 3 calls the framework to include some identification mechanism for CRDTs. This mechanism should allow identifying cohesive replicas for the same specific CRDT instance. As this requirement needs to be combined with requirements 2, 4, and 5, the framework needs a reliable naming scheme for replicas created in multiple nodes.

The four remaining distribution transparencies do not play a role in this thesis because of the Selective Hearing specification [35]. It specifies that each node maintains a copy of the replicated data types. The three transparencies *location*, *migration* and *relocation*, all deal with a hidden location of objects. As Selective Hearing only updates local replicas, different locations and object movements cannot be hidden. Additionally, Selective Hearing assumes that nodes fail by crashing and recover as if they were new nodes. This thesis follows this assumption and does not try to increase fault tolerance (for example, by introducing data persistence to prevent data loss). This decision limits the *failure* transparency. However, many typical sources for failures in distributed systems are avoided by using CRDTs and gossip protocols. In chapter 2, Selective Hearing, CRDTs and gossip protocols are discussed in more detail.

This work only covers parts of distributed systems' other goals [53]: resource sharing, openness, and scalability. The only resources this framework supports sharing among nodes are states

of CRDTs. With Computational CRDTs (C-CRDTs) (discussed in Section 2.2.2), these states are the results of computations. So, in some sense, this framework shares computational resources among the nodes. However, besides the replication of the CRDT states, no additional resources are shared by this framework. The goal for being open includes the design of platform-independent protocol specifications. As a result of not trying to be open, all messages used for communicating in the gossip protocols presume the receiver has the same software installed. Similarly, the CRDT states can only be synchronized between instances of this framework. The work required to design and test platform-independent messages and state distribution is beyond this project's scope and has been left for future projects. Open distributed systems should be extensible, which is an aspect that this framework should achieve by using object-oriented design patterns [13, 50]. The scalability in node size has been limited to smaller groups. In this proof-of-concept, there is no attempt to scale the group membership across different geographic locations. Additionally, the implementation has not been optimized for runtime performance, which also affects the scalability. Administrative scalability was also not considered. Each of these goals, although important for usage in production, will not hinder the objective: to investigate the potential for stable and reliable usage of Selective Hearing in the Loomo context.

## 1.3 Methods

A CRDT framework is designed based on the requirements mentioned in section 1.2. The framework has to be implemented in the Java programming language version 8 – the version supported by a Loomo running Android 5.1 (API Level 22).[17] Different parts of the project are encapsulated to create an object-oriented design. In this process, design patterns by the *Gang of Four* [13, 50] and principles such as SOLID [33, 34] are followed where applicable. Git[3] is the version control system used in this project. The GitLab[4] instance provided by the Hochschule für Angewandte Wissenschaften (HAW)[5] has been used to keep track of issues and work in progress through Merge Requests. A GitLab pipeline is set up to support continuous integration. The CRDT framework's source code is available under the MIT License [59] in the CaDS GitLab group.[6]

To evaluate the CRDT framework, a simple application that uses the framework is developed. This evaluation application is executed in a Mininet environment. During the evaluation execution, incoming and outgoing messages from the gossip protocol are logged. This log output is used to calculate multiple metrics for different parameters. There are no standard metrics to evaluate distributed systems. The metrics reliability, Relative Message Redundancy (RMR), and Last Delivery Hop (LDH) have been introduced in the Plumtree paper [27]. As every message

[3] https://git-scm.com/, accessed 19th May 2021
[4] https://about.gitlab.com/, accessed 19th May 2021
[5] Thanks to the team maintaining the instance!
[6] https://git.haw-hamburg.de/smart4cads/crdt4loomo/, accessed on the 21st April 2021

Table 1.1: Evaluations with their metrics and adapted parameters

| No. | Metric | Parameter |
|---|---|---|
| 1 | sent − received messages | host number; packet loss rate |
| 2 | reliability | host number |
| 3 | RMR | packet loss rate |
| 4 | LDH | host number |

sent by Selective Hearing is broadcast by the Plumtree protocol, these metrics can be used to evaluate this framework. The evaluations, their metrics and the parameters being evaluated are depicted in Table 1.1. Evaluation 1 shows the difference between sent and received messages in the log output. Due to the evaluation implementation, it is expected that every message is logged as sent before being logged as received. This metric's outcome is used to indicate whether the results of the evaluations that follow are credible. The second evaluation is used to show the reliability expected for different Loomo fleet sizes. Third, the RMR is investigated for different packet loss rates in the network. Finally, the LDH metric is compared in different fleet sizes. According to the Plumtree protocol specification, it should be possible to observe higher LDH values for larger fleets. The results of this evaluation could indicate errors in the protocol implementation.

## 1.4 Structure

Chapter 2 explains the foundational technologies used in this thesis. An overview of the Selective Hearing programming model is provided. This includes CRDT concepts, the hybrid gossip protocol Plumtree, and the gossip membership service HyParView. Additionally, this thesis is compared with related work. Chapter 3 presents the framework architecture that has been designed to address the requirements. An object-oriented design is constructed based on the algorithms mentioned in the previous chapter. Deviations from the original algorithms, the rationale behind their intentions, and implementation consequences are discussed in this chapter. Highlights and pitfalls from the implementation are pointed out in Chapter 4. These include the choices made for Netty in the HyParView implementation, generic CRDT implementations, and Java serialization instead of a custom format. Solutions for some issues described in the previous chapter are addressed. The built framework is evaluated in Chapter 5. Multiple tests are performed, and their results are observed. The test results and their impact on our motivational example involving a Loomo fleet are analyzed. The thesis concludes with Chapter 6. The research findings are summarized, and potential future projects are mentioned.

# 2 Background and Related Work

This chapter explains the foundational technologies used in this thesis. The distributed systems' context for the protocols is provided based on [53]. CRDTs [48], Plumtree [27], and the Hybrid Partial View (HyParView) [26], are relevant topics. All of these are used to construct the Selective Hearing programming model [35]. An overview of the model follows. Other CRDT frameworks are briefly mentioned before the chapter concludes.

## 2.1 Consistency and Availability

According to [53], a "distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system." These autonomous computing elements are referred to as "nodes." In Section 1.2, relevant distribution transparencies have been named. In distributed systems, in which the *failure* transparency is not achieved, failures can prevent the collection of nodes from appearing as a single coherent system.

One sort of failure which can occur in distributed systems is network partitions. A network is partitioned if a non-empty group of nodes is disconnected from another non-empty group of nodes in the network. Network partitions can be caused by various reasons, ranging from failing routers to sharks destroying undersea cables [14]. Software developers constructing distributed systems need to take network partitions into account and mitigate their effects on the application. In the case of network partitions, the developer's choice has been between consistency and availability. This is called the CAP theorem [15]. If the developer needs to guarantee consistency, the application cannot be available if a partition occurs. If the application were available, conflicting or incompatible changes to the application could occur in both partitions.

To increase failure transparency, some systems favor availability and reduce their consistency guarantees. Every node accepts changes to the system, but they might not be propagated immediately to every node in the system. Instead, all updates are distributed to every other node at some point in the future. This weaker form of consistency is called eventual consistency. Eventually consistent distributed applications can be available even when partitions occur. Updates to a partitioned application are propagated across the complete network if the partition has been resolved.

Distributed systems that guarantee eventual consistency need to replicate the application data. In contexts where conflicting updates can be prevented, there is no problem to achieving the *replication* transparency. However, in applications which are prone to produce write-write conflicts, additional mechanisms are required to make the replication transparent.

## 2.2 Conflict-free Replicated Data Types

A Conflict-free Replicated Data Type (CRDT) is a data type designed to be replicated and updated without coordination among multiple nodes. CRDTs have been introduced by [49] as an approach to achieve strong eventual consistency. Strong eventual consistency extends eventual consistency with the requirement that all replicas receiving the same set of updates reach the same state.[49] It has been argued that CRDTs have changed the "rules" of the CAP theorem in [8]. A comprehensive guide about the uses of CRDTs has been provided in [47]. A detailed CRDT overview is given by [44], which is aimed at developers using existing CRDT libraries, building systems to support CRDTs, or designing new CRDTs.

According to [44], the main properties of CRDTs are that every replica of a CRDT can be updated without coordination with other replicas. Two replicas deterministically reach the same result if they have received the same set of updates. This is known as the synchronization model and is discussed in Section 2.2.1. The concurrency semantics are discussed in Section 2.2.2 following the synchronization model. They describe how two replicas can deterministically reach the same result.

### 2.2.1 Synchronization Model

The synchronization model of CRDTs describes the data that is used to synchronize CRDTs between replicas.[44] A CRDT that has been specified in one model can be emulated in every model [47, 3]. Every model has its advantages and disadvantages.

#### State-based CRDTs

State-based CRDTs synchronize replicas by propagating their full state. Other replicas' states get joined into the local replica. The join operation of a state-based CRDT is associative, reflexive, and idempotent. Because of these properties, state-based CRDTs have very minimal network requirements. Updated states can arrive in any order. As long as every replica receives every update at least once, they reach the same state.

An additional advantage of the join operation is that state-based CRDTs do not need to be synchronized after every update. The state can instead be synchronized after multiple

updates to a replica. This feature can reduce bandwidth usage because states typically grow monotonically.

A disadvantage of state-based CRDTs is that they always require the entire state to be propagated, even if an individual update only affects a small part of the state. They are also used by Selective Hearing because of the low network requirements. All CRDTs implemented in this thesis are state-based CRDTs.

### Other Synchronization Models

**Operation-based CRDTs** synchronize replicas by propagating every update. In an operation-based CRDT, every update consists of two functions, a stateless generator function and an effector function with side effects. The replica receiving the update executes the generator function. The generator function does not change the replicas state. Instead, it generates the effector function based on the current state of this replica. This generated effector function is then propagated to every replica, including the one generating it. Once it has been executed, the effect of the update is present in the state.

Updates need to be designed in an idempotent manner, or the system must make sure that each update is received by every replica exactly-once. Otherwise, if an update is received and executed multiple times on some replicas, their state would no longer be deterministically the same. Compared with state-based CRDTs, operation-based CRDTs have the advantage of not requiring the entire state to be shipped if only one part has been changed. However, if most updates affect the whole state, operation-based CRDTs synchronize each update individually. In addition, operation-based CRDTs have high network requirements because the operations are not self-contained. Selective Hearing does not meet these requirements; therefore, this thesis does not use operation-based CRDTs.

**Delta-state CRDTs** synchronize replicas by only propagating changes to the state since the last synchronization. It has been verified by [3], that detla-state CRDTs also achieve strong eventual consistency. They were introduced by [1] and combine the advantages of state- and operation-based CRDTs. Delta-state CRDTs only synchronize the effect an update had on the state of a CRDT. So, as in operation-based CRDTs, they reduce bandwidth because they do not require the full state to be synchronized. They include a join operation similar to state-based CRDTs. The advantage here is that multiple updates can be piggybacked and do not need to be synchronized individually. However, as the delta-states are no longer self-contained, they require causal delivery. Selective Hearing uses a communication layer that does not guarantee any delivery order; therefore, delta-state CRDTs are not used in this thesis.

### 2.2.2 Concurrency Semantics

CRDTs can be designed with different concurrency semantics. A concurrency semantic describes the CRDT conflict resolution if concurrent updates occur. The concurrency semantic depends on the CRDT update operations. Thus, every data type has individual possible semantics. In this section, some concurrency semantics implemented in this thesis are explored.

#### Logical Time and Clocks

CRDTs need to be able to determine the order of updates. Time is a method to create such an order. However, physical time cannot be reliably used in distributed systems. A detailed discussion of the issues is beyond the scope of this section. A good overview of this issue can be found in [53]. Logical time is a standard approach to address the order of update operations.

Multiple logical clocks exist. These are algorithms used to determine logical time. Most of the approaches "translate back to a simple causal history."[2] To create a causal history, every operation receives a logical timestamp. This timestamp can have the form $(i, c)$, with $i$ being the unique identifier of the node initiating the operation, and $c$ being a counter of the operations on this node. This form ensures a unique timestamp per operation.

The causal history of an operation is a set containing timestamps of all previously observed operations. This set is attached to the operation, which can now be ordered. A node $n$ receiving an operation $o$ will only apply $o$ if it has applied all operations that are contained in $o$'s causal history. An excellent introduction into different versions of logical clocks and their relations to causal histories has been provided by [2].

In this project, vector clocks have been used in the implementation. Vector clocks derive from causal histories. They only keep the timestamp of the latest operation per node. This storage optimization works because the later timestamp indicates that all earlier operations have been available.

Additionally, as described in [11], hybrid logical clocks have been implemented in this project. These clocks are not related to causal histories. They are called "hybrid logical" because they combine a physical timestamp and a logical part to a timestamp. These hybrid logical timestamps do not guarantee causal consistency—as causal histories do. Instead, they can be used deterministically to decide which operation all nodes consider to be the latest operation.

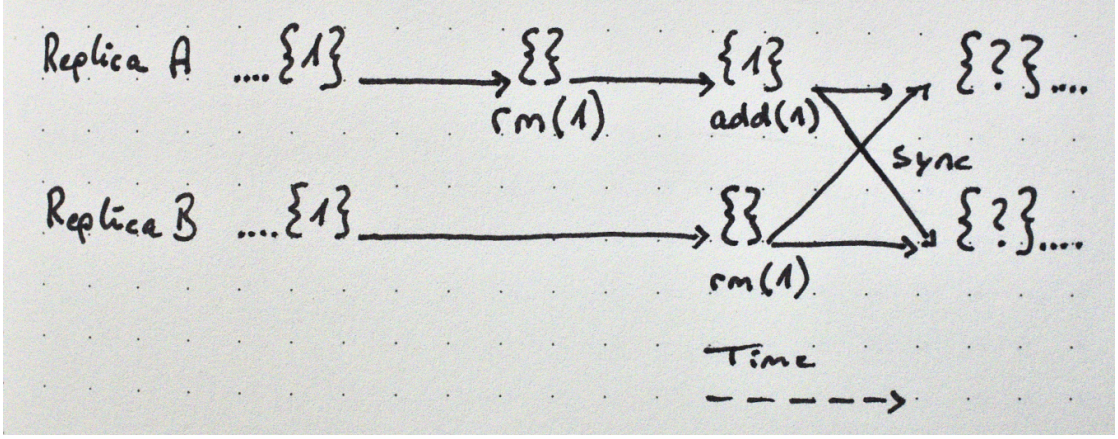


Figure 2.1: Set CRDT concurrently updated by two replicas with a conflict.

### Last-Write-Wins

The Last-Write-Wins (LWW) semantic is a concurrency semantic commonly used in multiple data types. As the name implies, two conflicting updates are resolved by using the latest timestamp in their total order. The LWW semantic is often achieved by using hybrid logical clocks, described in 2.2.2. In the register CRDT implemented by this thesis, the LWW semantic is used. A register CRDT holds a single value that can be set and retrieved.

In a LWW register, the value to be retrieved is the value that was written with the highest timestamp. The alternative register semantic is the multi-value register. This register returns a collection of all concurrently written elements.

The LWW semantic has been used to implement a set data type in this framework. In a set, elements can either be present or not; they do not have a specific order and cannot be added multiple times. Figure 2.1 shows a conflict situation in a set. Two replicas concurrently update a set using the same element. Replica A adds the element to the set, and Replica B removes the element from the set. In a set with the LWW semantic, the conflict with this element is resolved by the total order of the operations, based on their timestamp. Alternative set semantics give priority to one of the operations, resulting in an add-wins or remove-wins set.

### Computational CRDTs

The C-CRDT is described in [39] in the context of incremental stream processing. A C-CRDT performs a computation over the elements added to a CRDT replica. It does not maintain all individual elements added to the CRDT. Instead, only the result of the computation over the values is of interest. In [39], the application programmer is always required to provide “gluing

code" to connect multiple C-CRDTs. The authors expanded their study on C-CRDTs in [38] for transparent usage in cloud databases. They described three state-based computational design patterns that do not require any additional action from C-CRDTs' users. In this thesis, the first two design patterns have been used in implementations of the CRDT library. Generic algorithms describing the C-CRDTs implemented in this project are specified in [38].

**Incremental Computations Design** In this design, each replica maintains its contribution to the overall value. An updated replica propagates its full state to other replicas. The positive-negative counter and the average C-CRDT are implemented based on this design in this project.

In an average C-CRDT, each replica maintains two data points: first, the sum of all elements added to this replica; second, a count of the individual elements that have been added. The average value is calculated with the sum of all individual sums divided by the sum of all individual counts.

**Incremental Idempotent Computations Design** In this design, replicas only maintain the final result. Updates directly change the result, which is propagated to every other replica. A C-CRDT based on this design is a top-K list, in which elements that do not belong in the top-K at one point never belong to this list in the future. It might be used to implement a high score with $k$ ranks. The top-K list has been implemented in this framework.

**Partially Incremental Computations Design** An example C-CRDT based on this design is a top-K list, where elements that are at first not part of the top-K, can become relevant in the future. This could happen when a top-K element gets deleted, and a previously discarded element takes its place in the top-K. A top-K replica only synchronizes elements that are part of the top-K. However, the replica needs to maintain other elements so that they can later be propagated to the other replicas. C-CRDTs following this design have not been implemented in this proof-of-concept. As every replica could contain data that later becomes relevant, the replica should stay available in the future. Selective Hearing expects nodes to fail by crashing. Recovering nodes start from scratch as if they were new nodes. This condition could result in data loss with partially incremental C-CRDTs.

## 2.3 Gossip protocols

To appear as a single coherent system, the nodes in the collection need to communicate with each other. As described in Section 2.2, CRDTs require each replica to receive every update. Broadcast mechanisms can deliver a message to each node in a group. Different data dissemination patterns

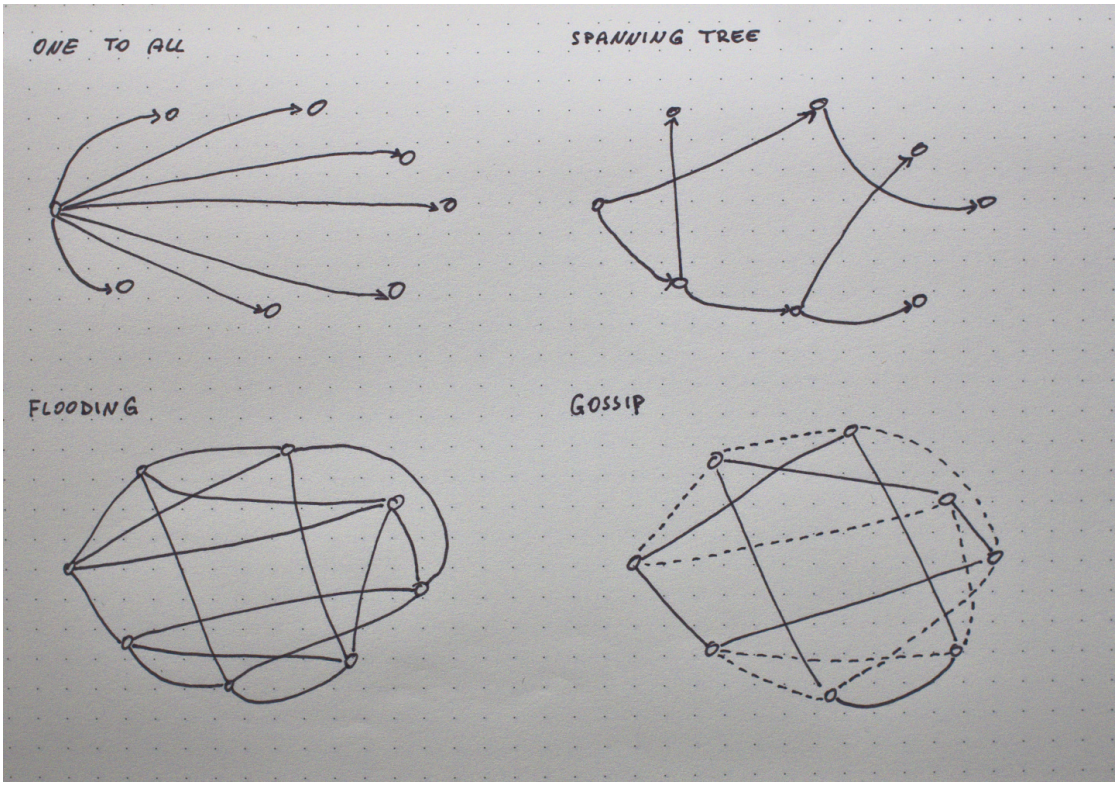


Figure 2.2: Comparison of the data dissemination patterns one-to-all, spanning tree, flooding, and gossip with eight nodes.

used in broadcast algorithms are discussed in section 2.3.1. Broadcast algorithms can also use different strategies to distribute data. Section 2.3.2 discusses the push and lazy-push strategies. Plumtree[27] combines two dissemination patterns with different push strategies, as is laid out in section 2.3.3. To deliver a message to every node in a group, Plumtree and other broadcast protocols need to know the member nodes of this group. HyParView has been used by [27, 35] as a membership protocol for Plumtree. A brief introduction to the HyParView algorithms is provided in Section 2.3.4.

### 2.3.1 Dissemination Patterns

Dissemination patterns describe the different ways data can be broadcast across a set of nodes. Their role in the construction of Plumtree has been presented in [28]. Figure 2.2 shows four patterns that will be laid out in this section.

**One-to-All** The one-to-all pattern is straightforward. The node with new data sends it to each node in the network. The nodes receive the message at most once, so message duplications do not need to be addressed. Although this pattern is easy to implement, there are some major drawbacks: It scales poorly, does not balance the load, and does not tolerate failures. Scaling is an issue here because each node needs to know about all the other nodes in the network. If there are many nodes, the membership management could have a massive impact on bandwidth, performance, and memory. Performance is also a problem when a single node sends a message to every other node in the network. The sending node has many messages to send, while every other node is idle. Additionally, if the sending node fails after half of the nodes have received the message, the remaining members never receive the broadcast. This can cause inconsistencies.

**Spanning Tree** The spanning tree pattern improves upon the scaling and performance issues of the one-to-all pattern. Every node sends the broadcast message to its children. Similar to the one-to-all pattern, the spanning tree pattern prevents message duplication. The load of reaching each node in the network is distributed among the members of the system. However, the load is not distributed equally among the members. Leaf nodes do not send messages to other nodes unless they are the origin of the broadcast. Nodes near the top of the tree need to send many more messages. In addition to load balancing, nodes only need to know about parts of the network membership. This allows for more extensive networks without having each node maintain membership data for every other node. However, the spanning tree pattern is less straightforward to implement because it needs to be constructed covering all network members. In addition, just like the one-to-all pattern, it is still prone to failures. If a single node fails before sending the message to its child nodes, the broadcast does not reach every active node.

**Flooding** The flooding mechanism introduces redundancy in message delivery to achieve higher resilience to failures. Each node has a specific number of neighbor nodes to which it forwards received broadcast messages. Failing nodes no longer cause a broadcast message to be lost. A drawback of this approach is that nodes receive single broadcasts multiple times and need to deal with message duplicates. For example, it is important that nodes only forward messages when they received them the first time, to prevent endless broadcasts. Another drawback is higher bandwidth usage. On the plus-side is the load distribution; every node contributes an equal number of messages per broadcast. Implementing this pattern is less complex than a spanning tree construction.

**Gossip** The gossip pattern is an adaption of flooding. It builds on the strengths of the flooding pattern while improving upon some of its drawbacks. Message redundancy is still used to mitigate the effect of failures. However, the message redundancy is reduced. To cut down on message duplication, a node in a gossip protocol sends its messages to a random subset of its neighbors.

This still covers every node with a high-reliability rate. However, the random approach makes this pattern less predictable than the other patterns.

### 2.3.2 Gossip Push Strategies

Different strategies for broadcasting data with a gossip protocol exist. The differences lie in the responsibilities of the sending and receiving nodes. Although more strategies exist, the two relevant for Plumtree will be explored in this section.

**Eager Push** In this strategy, a node sends the entire message to its neighbors. As soon as a recipient node knows a certain message exists, it also has the message's payload and can deliver it. Thus, the implementation of the receiving process on the recipient node is rather simple, as it only needs to make sure that duplicate messages are detected. The eager push strategy reduces the latency from sending a broadcast until all nodes have received the broadcast. The cost of this low latency is higher bandwidth usage, as the full payload is transmitted with each message.

**Lazy Push** In this strategy, a node only sends a message announcement to its neighbors. The neighbors can request the entire message if the full payload is needed. It takes three round-trip-times, with this strategy, for until a node to receive the whole broadcast message. This higher latency reduces the overall bandwidth usage, as the message announcements can be tiny. In addition to higher latency, the nodes are slightly more complicated when compared with eager push nodes. They need to keep track of received messages, message announcements, and the announcements' origin.

### 2.3.3 Plumtree

Plumtree is a hybrid gossip protocol introduced in [27]. A hybrid gossip protocol combines a gossip protocol with another distributed algorithm. The goal is to achieve the good qualities of both. "Plumtree" stands for "**p**ush-**l**azy-p**u**sh **m**ulticast **tree**." This name hints at the phases used in the Plumtree protocol: Plumtree performs an eager push in the spanning tree pattern, followed by a lazy push in the gossip pattern.

#### Plumtree Phases

**Eager Push Phase** In the first phase, an eager push is performed in the spanning tree pattern. The low-latency push strategy with higher bandwidth usage is combined with the low-overhead dissemination pattern. The spanning tree pattern mitigates the effect of the higher

bandwidth potential of the eager push strategy. When no failures occur in the network, this phase will propagate a broadcast to every node with low latency. If a node receives a payload message twice, it will request that the duplicate eager push connection be pruned by sending a prune message. The future message will be announced in the lazy push phase.

**Lazy Push Phase** In the second phase, a lazy push is performed in the gossip pattern. The low bandwidth push strategy with higher latency is combined with the redundant dissemination pattern. This serves two purposes: a) it allows the protocol to repair failures in the spanning tree, and b) the effect of failures can be reduced. After a node has pushed the payload message to its children in the spanning tree, it pushes an announcement of this message to each node in its direct neighborhood. If a node receives a message announcement first, it will wait until a timeout occurs and then request the payload from the announcement node by sending a graft message. If a message is requested, this will also trigger a tree repair process. The requesting node will receive the next message in the eager push phase.

### Plumtree Improvements

The Plumtree protocol already improves upon the gossip and spanning tree dissemination patterns. Its design still has limitations. Some improvements to these limitations have already been addressed [27, 28] and are briefly listed below. None of these improvements have been implemented in this proof-of-concept. However, the framework has been designed to allow improvements in future work.

**Shared and Sender-Based Plumtree** A straightforward Plumtree implementation shares a single tree for every broadcast. The authors thus call it "shared Plumtree" in [27]. As described in the spanning tree pattern on page 13, this single tree does not balance the load equally among all nodes. Additionally, a message sent by a leaf node is first sent up to the spanning tree root before it reaches nodes in other subtrees.

The "sender-based Plumtree" is an approach to mitigate these effects. Instead of using a shared spanning tree, the sender-based Plumtree constructs one spanning tree per sending node in the network. This method creates an optimal broadcast path for every sending node in the eager push phase. The drawback of a sender-based Plumtree is higher memory demands. This overhead can be minimized by lazy construction.

**Outstanding Set for Announcements** In [28, slide 131]. an improvement that uses an outstanding set for message announcements has been introduced. The tree repair process in the lazy push phase requires the announcements to arrive. If a message announcement is lost, the repair process might be hindered. To prevent announcement loss, a node adds

outgoing announcements to an outstanding set. A recipient either acknowledges the receipt of an announcement or requests the payload message. In both cases, the announcement sender removes the announcement from the outstanding set. Otherwise, the sender knows that the announcement has been lost and can resend it.

### 2.3.4 HyParView

HyParView is a hybrid gossip algorithm that provides a resilient membership protocol by using partial views [26]. It aims to provide global system connectivity in a scalable way. The used partial views ensure scalability. Each node only knows about part of the network. No node needs to know about all other nodes.

In HyParView, each node maintains an active and a passive view. The active view is used for data dissemination and failure detection. An active view relation is symmetrical. If node $A \in B.activeView$ is true, then $B \in A.activeView$ also needs to be true.[1] A node establishes a Transmission Control Protocol (TCP) connection to each node in its active view. The TCP connection allows for quick detection of disconnected nodes. The passive view is used to replace disconnected or failed nodes from the active view. No TCP connection is established to the nodes in the passive view. However, HyParView implementations may choose to establish TCP connections between nodes in the passive view. This increases the recovery performance if failures in the active view occur.

A new node sends a join request to a join node that is already part of the network to connect to the existing HyParView group. This join node adds the new node to its active view. If the maximum size of the active view has been reached, the join node drops an existing node from its view. It then forwards the join request to a different node in its active view. The forwarded message contains two time-to-live (TTL) values—one for the active view and another for the passive view. If the active-TTL is 0, the receiving node adds the new node to its own active view. If the active view of the recipient node is full, it drops an existing node from its set. However, receiving nodes can establish a connection to the new node if their active view is not filled. Receiving nodes perform similar steps with the passive view and the passive-TTL.

Periodically, nodes perform a shuffle mechanism. This mechanism's intention is to ensure that disconnected nodes get pruned from the passive views. A node initiates a shuffle, creating a list of nodes including itself, some nodes from its active view, and some nodes from its passive view. It sends this list to one of the nodes in its active view in a shuffle request. They forward the request similar to a join request. Once the shuffle-TTL is 0, the receiving node sends a shuffle reply to the initiator. This reply includes a nodes list constructed in the same way as the initial nodes list. Both of the shuffle request parties add the received nodes to their passive view. If

[1] As latency is never zero [53, page 24], in the real-world this condition is temporally false, if a new connection is established.

the list already contains the maximum number of nodes, they drop existing nodes, opting for the nodes they have sent in the shuffle message to the other node. Due to the construction of the node list—which includes the sending and active view—, running nodes are preferred in the shuffle process. Disconnected nodes get pruned from the system after some time elapsed.

## 2.4 Selective Hearing

Selective Hearing is a programming model for large-scale edge computing and is presented in [35]. This programming model consists of two layers: a programming layer and a communication layer that uses Plumtree. The programming layer is based on the Lasp programming language, which provides deterministic coordination-free computation based on CRDTs. The CRDTs are maintained in a node state, which is explained in Section 2.4.1. The operations in the programming layer are described in Section 2.4.2. They use the communication layer based on Plumtree to broadcast message to all nodes in the group (including the sending node).

### 2.4.1 Node State

In Selective Hearing, nodes maintain a state consisting of $(\sigma, \delta_I, \delta_v)$.[2] $\sigma$ is the known variables set, containing all variable identifiers observed by the node. $\delta_v$ is the known values set, similar to a map data structure. It contains $(i, v)$ tuples, where $i$ is an identifier and $v$ is the latest value received by the node. $\delta_I$ is the interest set for all pending *read* operations on the node. It contains a set of $(p, c)$ tuples for each variable identifier, added by the *read* operation. $p$ and $c$ both are functions with a variable value as a single parameter. Both functions are provided to a call of the *read* operation.

The Selective Hearing state is not persistent. The programming model expects nodes to fail by crashing. Recovering nodes choose a new identifier and start with a clean state. This can produce data loss in situations in which not everything has been propagated, see the C-CRDTs discussion in 2.2.2

### 2.4.2 Operations

A node can *declare* a new variable to create an identifier, *read* the value of a variable, and *bind* new values to identifiers. The specification for these operations has been made in a functional programming paradigm. In Section 3.4.2, an object-oriented design is presented, and differentiations from the original specifications are discussed.

[2] [35] uses $\delta_i$ instead of $\delta_I$, but this might become confusing with the identifier notation $i$.

$declare(t) \mapsto i$ *declare* creates a variable identifier $i$ on the initiating node. This identifier is broadcast to all the nodes through the gossip protocol. The identifier is connected to a type $t$. Each node receiving this identifier adds it to the known variables set $\sigma$.

$read(i, p, c)$ $i$ is the identifier of the variable whose value $v$ should be read. $p$ is a predicate that needs to be satisfied before a read operation is continued. In the case where $p(v) \mapsto true$, the continuation function $c$ is executed. If $p$ is not satisfied, the tuple $(p, c)$ is added to the interest set $\delta_I$. *read* does not perform a broadcast.

$bind(i, v)$ $i$ is the identifier of the variable whose value should be updated with the new value $v$. *bind* only broadcasts $v$ using the communication layer on the initiating node. Each node receiving this *bind* operation joins the received $v$ with the current entry in the known values set $\delta_v$. The result of this join replaces the previous value of $i$ in $\delta_v$. If no entry exists for $i$ in the $\delta_v$, a new tuple $(i, v)$ is added. After $\delta_v$ has been updated, all pending read operations on $i$ in the interest set $\delta_I$ are rechecked. If their predicate is satisfied after the update, the matching continuation is executed.

## 2.5 Other CRDT Frameworks

CRDTs are part of Atomix, "a reactive Java framework for building fault-tolerant distributed systems."[10] Atomix provides many algorithms for distributed systems, including the consensus protocol Raft, a multi-primary protocol, cluster management and communication, a standalone agent, as well as others. "Distributed Primitives" have been implemented using the various protocols. One of these methods is a CRDT protocol that uses a gossip protocol for network communication. Atomix is thus very similar to the design implemented in this thesis.

Additional CRDTs or CRDT frameworks have been implemented in different programming languages.[3] Two notable frameworks are Automerge [19] and Yjs [18, 41]. Both are implemented in JavaScript. Automerge is a JavaScript Object Notation (JSON)-like CRDT based on [20]. Yjs is a high-performance CRDT implementation that scales well when there are many users. It can be integrated into multiple existing text editors, which are frequently used on websites. Both implementations do not provide network communication. Instead, they communicate using existing peer-to-peer communication protocols. Compared with the proof-of-concept scope, other CRDT frameworks are more feature-rich. Porting one of these to Java for use on a Loomo is beyond this project's scope.

[3] An updated list is maintained at https://crdt.tech/implementations, accessed 11th May 2021.

## 2.6 Conclusion

To achieve higher availability in distributed systems, data objects are replicated among multiple nodes in a group. Updates to these objects cannot be propagated immediately to every node. Eventually, all nodes receive every update. This is known as eventual consistency. CRDTs have an additional guarantee: once all updates are received by every node, each node reaches the same result, with requiring coordination. This is called strong eventual consistency. Broadcast protocols can be used to propagate all updates to every node in a group. Different data dissemination patterns exist for broadcast protocols. The Plumtree protocol combines the fast but error-prone spanning tree pattern with the reliable but redundant gossip pattern. Plumtree uses HyParView to build the broadcast's membership group. The Selective Hearing programming model combines a programming layer based on CRDTs with a communication layer based on Plumtree.

The following chapter presents the solution strategy that has been chosen to implement Selective Hearing in this thesis. The object-oriented design is constructed based on the algorithms mentioned in this chapter. Deviations from the original algorithms, the rationale behind their intentions, and implementation consequences are also discussed.

# 3 Framework Architecture

This chapter presents the framework architecture that has been designed to address the requirements of this thesis. The object-oriented design is constructed based on the algorithms mentioned in the previous chapter. An overview is provided, and the three main components are laid out in more detail. Deviations from the original Selective Hearing algorithms, the rationale behind their intentions, and their consequences are discussed.

## 3.1 Architecture Overview

The framework architecture can be categorized using the architecture styles described by [53]. Although an application developer only uses objects, the architecture differs from a Remote Procedure Call (RPC) or Remote Method Invocation (RMI) architecture. The local object is not a stub, but completely functional and independent of remote objects. Selective Hearing resembles an event-based publish-subscribe architecture, which communicates using the broadcast protocols. Each node subscribes to every publication. The framework consists of four layers: an application, Selective Hearing, Plumtree and HyParView.

Figure 3.1 shows the high-level component diagram of the framework, which has been named 'crdt4loomo' in the Java package. For clarity purposes, this diagram does not include every detail of the components. The CRDT library and Selective Hearing are top-level packages in the framework. Another package hierarchy for the broadcast mechanisms is introduced. This package currently only includes a Plumtree and a HyParView implementation. The six most significant interfaces in this framework are depicted in this diagram. Their intentions are briefly explained in this section and laid out in more detail in the following sections. In most cases, an application developer accesses the interfaces defined by the CRDT library.

The motivation behind this design is to separate data types from communication. An application developer does not need to know which CRDT implementation is used. Components are loosely coupled and work with dependency injection [12]. The dependency inversion principle [30, 29] is respected because all components depend on abstractions.

The CRDT library defines interfaces of all supported CRDTs. Each supported CRDT is additionally implemented as a plain Java object using only the Java standard library. The

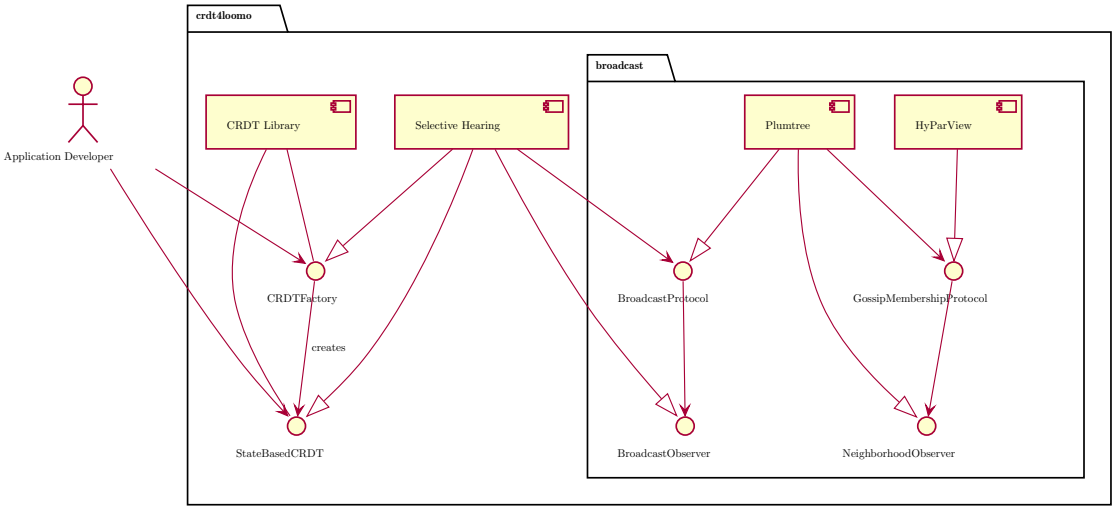


Figure 3.1: High-level component diagram on the framework.

state-based CRDT and CRDT factory are important interfaces of the library. The state-based CRDT defines the basic methods that every state-based CRDT in this library has to implement. It is used as the base interface for the following typed CRDTs. The CRDT factory defines the methods required to create instances of implemented CRDTs. This interface is implemented in the library and in Selective Hearings wrapped CRDTs.

Selective Hearing manages CRDTs that are propagated among nodes using gossip protocols. This component implements the state-based CRDT interfaces from the CRDT library. It also makes direct use of the CRDT library implementations but wraps these instances, adding the gossip propagation.

The broadcast package is the home for all components dealing with gossip protocols. No broadcast component knows what data it sends or receives. The gossip protocols implemented in this package belong in two categories: broadcast and membership protocols. The broadcast protocol interface defines how this package accepts messages for broadcasts from components using this protocol. Components using these broadcasts also need to receive broadcasts. This need is supported by the broadcast observer interface. This is an abstraction that the protocol implementation and caller can rely on. The gossip membership protocol interface defines the application programming interface (API) for a membership protocol. Implementations can be queried for all peers in the neighborhood. Users of a membership protocol can implement the neighborhood observer interface and register themselves to allow the users to be notified about updates.

The Plumtree component implements the Plumtree algorithms and satisfies the broadcast protocol interface. It forwards received broadcasts to every broadcast observer, retrieves the peers using a gossip membership protocol, and registers itself as neighborhood observer

to get updates on changes to its connected neighbors. Additionally, the Plumtree component implements the broadcast observer to be notified of broadcasts received by the gossip membership protocol. For clarity purposes, this implementation detail has been left out of the overview diagram.

The HyParView component implements the HyParView algorithms and satisfies the gossip membership protocol interface. It maintains an active view and a passive view of peers. Changes to the active peers are reported to all registered neighborhood observer instances. Additionally, the HyParView component also implements the broadcast protocol to notify its uses in higher layers of received broadcasts. For clarity purposes, this implementation detail has been left out of the overview diagram.

## 3.2 CRDT Library

Figure 3.2 presents the structure in the CRDT library. It shows the central interfaces and examples of how they are used inside the package. Two classes in the diagram have a lighter color than the rest. These classes are not directly accessible to users from outside the package. Application developers using this library only access the CRDT factory implementation of this library. Apart from this class, an application developer can only interact with CRDT interfaces. This ensures that an application using the CRDTs does not depend on a specific implementation. Still, if a developer really tries to enforce the use of local CRDT implementations, the application can depend on the local CRDT factory class.

Every CRDT in this library implements the state-based CRDT interface. It defines the two required methods that are always expected. One is the method to merge another CRDT. Additionally, the method defined by the comparable interface is required to allow the ordering of CRDTs.

Additional interfaces define the specific CRDTs, such as the set CRDT or an average CRDT. These interfaces extend the state-based CRDT interface and only have to add methods required for their semantic. In cases where interfaces in the Java API [43] exist, they are extended by the CRDT interface in this library.

### 3.2.1 Creational Pattern

Figure 3.2 also shows the CRDT factory. The factory interface is defined to always return interfaces instead of concrete classes. The local CRDT factory is the only class, which is directly accessible from other packages. It instantiates the local CRDT classes.

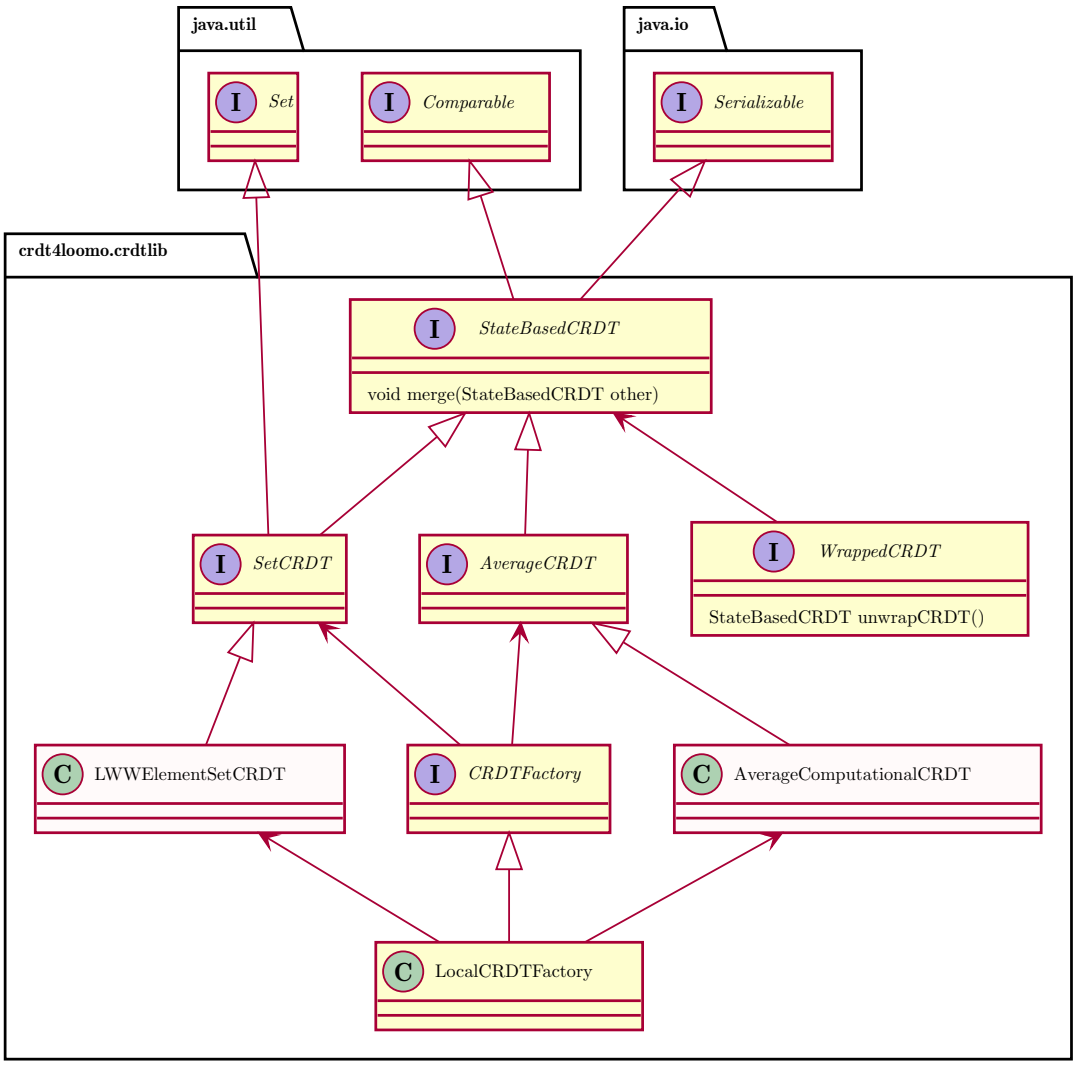


Figure 3.2: Interfaces used in the CRDT library.

In the decision on the creational design pattern used by the CRDT library, three options have been evaluated based on [50]: the factory method, the abstract factory, and the builder pattern. The abstract factory has been chosen to be used.

**The factory method** provides an interface defining how an object is created. Implementations of this factory method can decide which concrete object implementation is created. If used in this framework, this approach would lead to many different factory method interfaces and implementations: one for every CRDT interface. Many interfaces could become a source of errors when dealing with Selective Hearing, which requires that every Selective Hearing CRDT receives the same Selective Hearing instance. The use of multiple Selective Hearing instances could work, but it has not been tested. At the very least, it would waste computational resources. Additionally, the factory methods of different CRDT implementations could be mixed, leading to unexpected behavior. On the plus side, applications only have to depend on the few CRDT they are actually using. This would satisfy the interface segregation principle [31], which states that 'clients should not be forced to implement methods they do not use.' However, this could lead to applications that violate the information hiding principle because their inner state becomes part of their creation. In an implementation, all factory methods would probably be collected in a single class, which basically represents the next option.

**The abstract factory** provides an interface defining how a family of related objects is created without specifying what the concrete implementations are. An abstract factory creates products from product families. This option is appealing in this framework. The products are the concrete CRDTs. The product families are the packages that implement the CRDT interfaces. In this framework these are the local CRDT library and Selective Hearing. The CRDT factory interface becomes a part of the main CRDT library. This package also provides a local CRDT factory implementation that produces all the local CRDTs. The Selective Hearing component implements the factory interface as well. Every returned CRDT from an implemented abstract factory matches the other CRDTs from the same factory. One thing to be aware of is that a client of the framework might not require the usage of CRDTs from the same family. However, this is a minor issue, as the client can make use of different factories. It could be argued that an abstract factory violates the interface segregation principle [31]. Not every CRDT library might implement each CRDT interface, but they are required to implement every creation method. This was considered to be a minor issue in the scope of this project.

**The builder pattern** provides an interface that defines steps to create a single object with lots of options and, when finished, return the configured instance. This pattern could be used to create the different CRDTs as well. A CRDT builder interface could have methods to create CRDTs that manage a single element, multiple ordered elements or multiple unordered elements. Based on the method calls a register CRDT, list CRDT, or set CRDT could be returned. A

CRDT director could be introduced to simplify and hide the creation of concrete interfaces. An issue with a CRDT builder is that there is no CRDT implementation for every possible combination of options. As an example, the current state of the framework lacks a simple list. It would not make sense to use the builder instead of the director. The director would look very similar to an abstract CRDT factory, which could be used instead.

### 3.2.2 Wrapped CRDTs

It is not a general rule that CRDTs from one factory can be combined with CRDTs from another factory. The merge operation only allows the merging of CRDTs from the same classes. This limitation is necessary because the merge operation needs to access the internal CRDT data structure, which is not accessible using the interface. The underlying data structure might become inconsistent with different CRDT implementations.

A CRDT package, such as Selective Hearing, does not have to implement the CRDTs on its own. Instead, it can follow the do not repeat yourself (DRY [54, 58]) principle and rely on the implementations in the local CRDT library for tested CRDT implementations. These implementations are wrapped with additional functionality. The wrapped CRDT interface provides a way to access an underlying CRDT implementation. This access ensures the interoperability of CRDTs from different CRDT factories, such as Selective Hearing, with the local CRDTs. Before comparing the classes for compatibility, the merge operation unwraps a CRDT until the actual implementation is returned.

An application use case for wrapped CRDTs is to utilize them to batch multiple consecutive updates to a CRDT. The local CRDT can be updated in a loop multiple times in a row. After all consecutive changes have been applied, the local CRDT can be merged into the Selective Hearing CRDT, which then propagates all changes once instead of every change on its own. This increases memory usage but can drastically reduce network traffic, especially with state-based CRDTs, which always propagate their full state. This scenario might seem to contradict the distribution transparency. This is not necessarily the case. An application could accept two CRDT factory instances – one for immediate updates and another for occasional updates. The application does not need to know which implementations are used or if different factories are used at all.

Instead of using a separate wrapped CRDT interface, the unwrap method could have been added to the default state-based CRDT interface. This option would have been more flexible. The unwrap method of CRDTs that wrap another CRDT returns the result of the unwrap method of the wrapped CRDT object. A CRDT that does not rely on another CRDT just has to return the reference of itself (`this` in Java). This approach would remove checks if an object is an instance of a wrapped CRDT interface. It would also ensure that the unwrap method always returns an unwrapped CRDT.

An alternative to using a wrapped CRDT interface could be a data transfer object that defines the format used to exchange data between two operations. However, it might not be possible to create a useful variant of such a data format that satisfies different CRDT semantics. Designing such a format is beyound the scope of this project.

## 3.3 Broadcast Package

The broadcast package defines multiple interfaces, depicted in Figure 3.3. These interfaces are used and implemented by different components in the broadcast package. The components and their internal relationship are presented in Figure 3.4.

### 3.3.1 Broadcast Package Interfaces

The peer interface in Figure 3.3 represents one neighbor of the current node. All peer objects should be uniquely identifiable and support sending messages to the node they represent. Note that this interface is technology-agnostic. The same interface could be used for TCP and Bluetooth connections without requiring the calling party to deal with the connection details. The calling party can send a message, and the peer knows how to do it. The name parameter in the message can be used by broadcast observers to only handle their messages.

The broadcast protocol interface allows low coupling between the broadcast component and its users, such as the Selective Hearing component. This design satisfies the dependency inversion principle [30, 29] because the high-level module Selective Hearing and the low-level module Plumtree depend on the same abstraction.

In addition to sending messages through the broadcast protocol, components need to be able to receive broadcasts. Users of this broadcast component can implement the broadcast observer. An observer is required to be registered in advance in the broadcast protocol. Messages received before an observer is registered are lost. This was considered an insignificant issue because the construction of the protocols allow the registration of observers before the protocol algorithms are executed. Using the observer pattern would also allow multiple receiving components to use the same broadcast mechanism.

Two ideas have been cast off in the broadcast observer interface's design process. Firstly, a simple queue for received messages could have been used. The broadcast protocol could just append messages to the queue and observing components could access the messages from there. The advantage here is that messages received before an observer is registered are not lost. However, it would have complicated concurrent usage of multiple receiving components. Additionally, queue operations would have made the broadcast protocol interface more complicated, which was undesired in this proof-of-concept. Secondly, an object implementing a step in the network

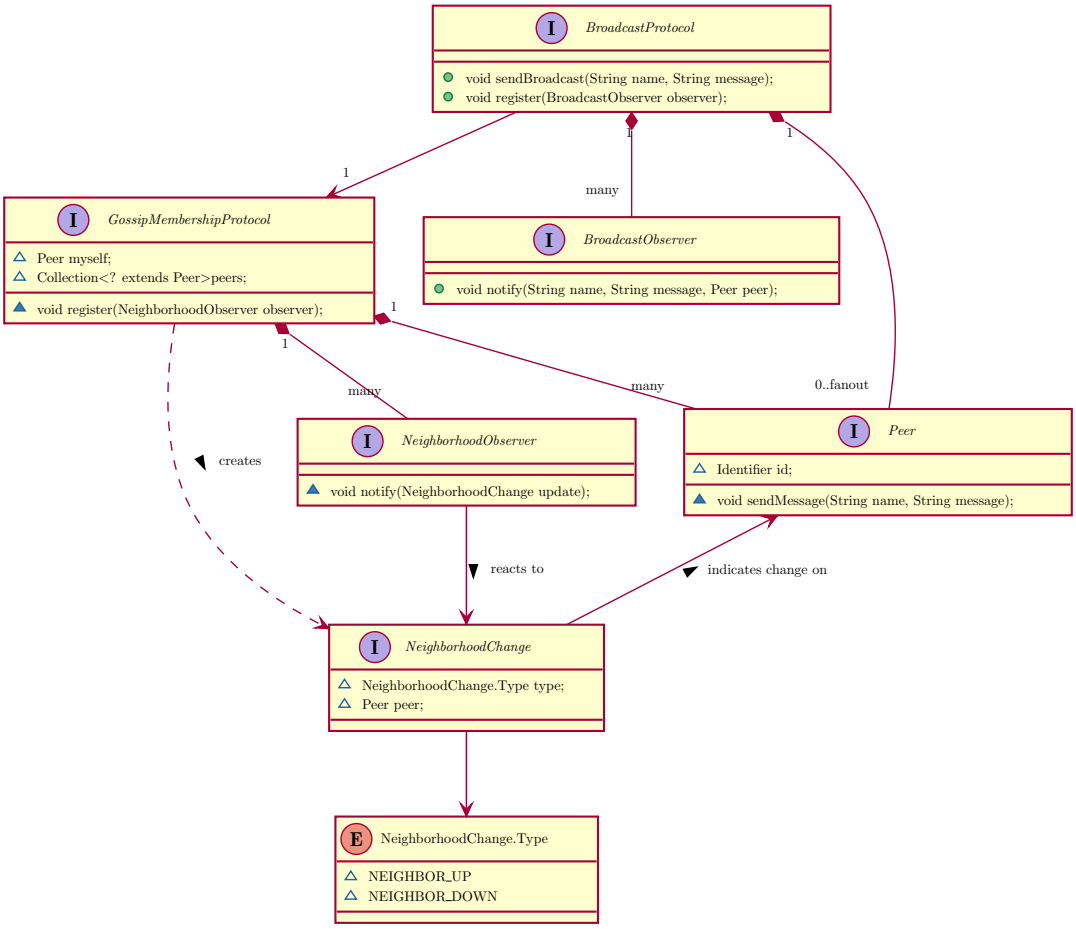


Figure 3.3: Interfaces used in the broadcast component.

communication framework's receiving pipeline could have been used. This would have reduced the required conversion steps between the components. However, it would have exposed the broadcast mechanism's internal dependencies. Moreover, it would have forced the user to add the network framework as dependency as well.

Lower layers should not need to care what type of data they are propagating. So, in this framework, a text representation of the propagated data has been used. In future work, more sophisticated methods to exchange data might be used.

The interfaces are designed to support effortless testing. When using dependency injection [12], the Plumtree protocol does not have to know whether it is dealing with real nodes in a network or stub implementations used for testing.

Plumtree maintains two peer sets. To initialize the eager push peers set, the protocol needs to receive its peers from the membership protocol. These sets need to be updated when the neighborhood or membership status of a contained peer changes. In [27], two primitives indicating a neighbor is down or up have been specified. The gossip membership service needs to trigger the events "to notify the gossip protocol whenever a change happens on the partial view maintained by the peer sampling service." These primitives have been implemented with the observer pattern again to achieve low coupling between the broadcast and the membership protocol. The neighborhood change can be observed. It has a type (either up or down) and affects a peer. The broadcast protocol implements the neighborhood observer interface and can be registered in the gossip membership protocol. Note that although the membership protocol reports the changes, the interfaces use 'neighbor' or 'neighborhood' in their names and not 'membership.' This is because the changes are only reported about peers in the current node's neighborhood. The neighbor may have connected to or disconnected from the current node. However, this might not always affect the membership itself. Consider the following scenario: A new node connects for the first time and is added as a peer to a random node. This process triggers a neighbor up event on the random node. It also indicates a membership change because the node is new in the group. The randomly selected existing node might have to drop a neighbor from its peer set due to limits on the active view size. This triggers a neighbor down event on both nodes. It does not indicate a change in the membership, even though the neighborhood for both nodes changed.

### 3.3.2 Broadcast Implementations

Figure 3.4 shows how the concrete protocol implementations Plumtree and HyParView use the broadcast package interfaces.

HyParView is a gossip membership protocol. It implements the peers that are maintained and how the changes are represented. It notifies a neighborhood observer about changes to its peers.

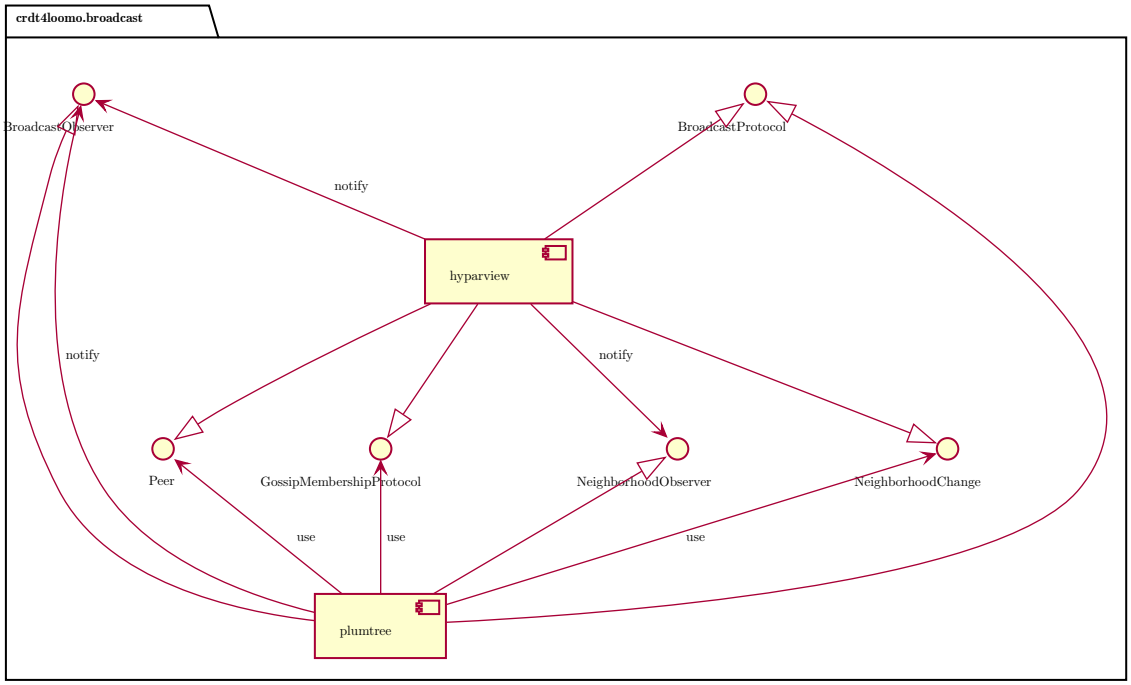


Figure 3.4: Broadcast component diagram.

It also performs and receives broadcasts and notifies broadcast observers about messages it received from a peer.

Plumtree is a broadcast protocol. When receiving a broadcast, it notifies the broadcast observers that have been registered. It relies on a gossip membership protocol to receive peers as neighbors. It observes its neighborhood for changes on neighboring peers.

In addition to the interfaces, which are all defined directly in the broadcast package, only a single facade for each of the components has to be accessed from outside the package.

## 3.4 Selective Hearing Component

The most important classes and interfaces of the Selective Hearing component are shown in Figure 3.4. The Selective Hearing interface and its implementation are central in this package. For clarity purposes, together they are referred to as Selective Hearing class in this section. The Selective Hearing class implements the state and operations, as described later in this section. It implements the state and operations, as described in Section 3.4.2. This class maintains all CRDT states without knowing the concrete CRDTs being used. This is achieved using the VariableValue interface. Each CRDT implemented by the Selective Hearing middleware also

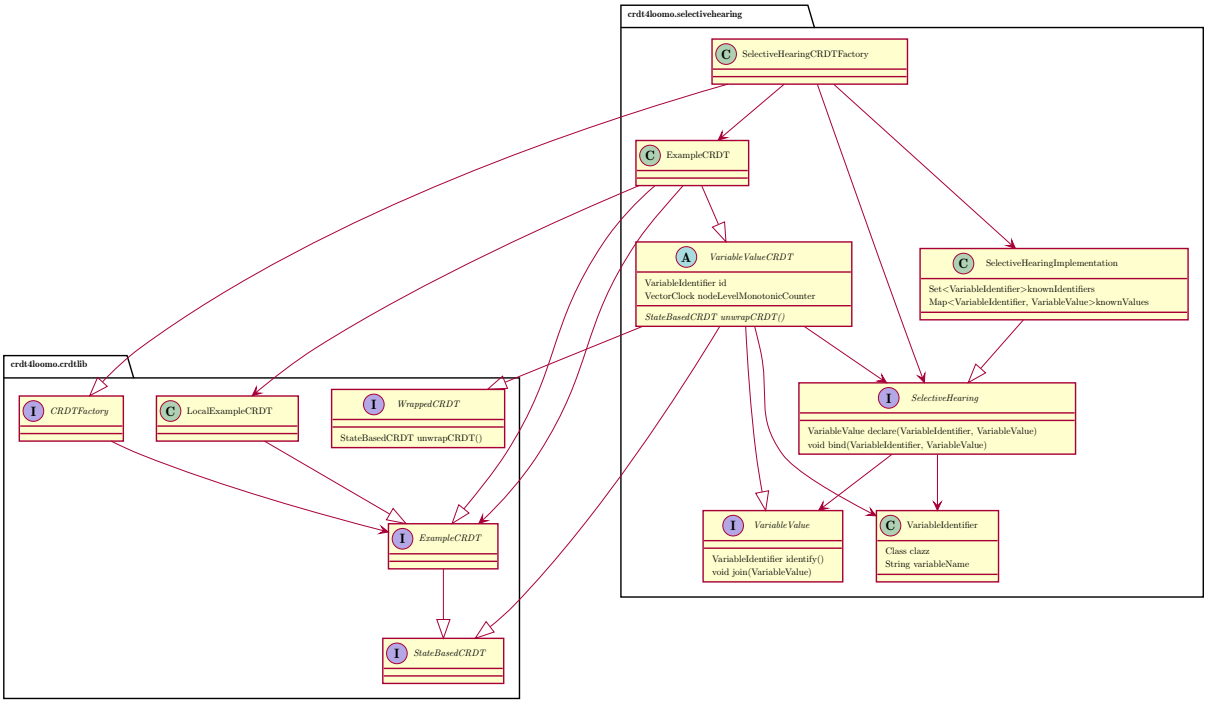


Figure 3.5: Interfaces and packages of the Selective Hearing component.

implements the VariableValue interface. This is the abstraction that the Selective Hearing class and the CRDT depend upon. This design follows the dependency inversion principle [30, 29]. The abstract class VariableValueCRDT implementing the VariableValue interface removes duplicate code among the CRDTs. It does so by utilizing the WrappedCRDT interface to access the local CRDT consistently.

The user of this package only has to access the Selective Hearing CRDT factory and the interfaces defined by the CRDT library. The factory knows how to construct the Selective Hearing class and all CRDTs in the package. Currently, a Selective Hearing CRDT instantiates the local CRDT it wraps. This instantiation can be seen in pseudocode later in the section. This approach has been a consequence of technical decisions made in the identifier implementation. The disadvantage is that the local CRDT implementation is fixed. This design violates the open-closed principle [37, 32]. For example, consider a persistent CRDT library that wraps local CRDTs just as Selective Hearing does. If this library were used in the Selective Hearing middleware, all the Selective Hearing CRDTs would have to be changed. A more sophisticated approach would use dependency injection [12]. The Selective Hearing CRDT factory could be optionally initialized with a CRDT factory parameter that is used to construct the wrapped CRDTs. This factory could be passed into the Selective Hearing CRDTs.

As depicted in Figure 3.1, the Selective Hearing class acts as a broadcast observer and accesses a broadcast protocol. These connections have been omitted from Figure 3.5 for clarity. Additionally, the interfaces and implementations used from the core package, such as the identifier interface or the vector clock implementation, have been removed.

### 3.4.1 An Example CRDT in Selective Hearing

In Selective Hearing, the CRDT library interfaces are implemented. The Selective Hearing implementations of these CRDT interfaces wrap the existing local implementations. The pseudocode of a wrapped example CRDT is presented in Algorithm 1. This pseudocode implements a fictional state-based CRDT, consisting of an update, a query, and a merge operation. The wrapping CRDT also implements a join operation which is part of the variable value interface that the Selective Hearing class manages.

In Algorithm 1, the bind operation can be observed in action whenever the state of the CRDT has changed. This is the case after an update or merge operation. However, bind is not called after a received example CRDT is joined. This would lead to an infinite loop of bind messages. All operations in this wrapping CRDT simply forward the calls to the local CRDT and execute very few other operations.

The declare operation is never used in Algorithm 1. Due to the signature of the declare operation in Algorithm 4, declare is called from outside after the CRDT has been completely initialized. The call is performed in the Selective Hearing CRDT factory as demonstrated in Algorithm 2.

### 3.4.2 Functional Logic to Object-Oriented Design

The Selective Hearing operations have a mathematical specification in [35]. The operations are specified in the context of a functional programming environment. Variables are accessed directly by name using the read operation in a recursive function (the example in [35, Figure 3] is a filter).

One goal of this thesis is to provide a CRDT library to an application programmer. Another goal is to make the distribution transparent. The programmer should not need to adapt the code depending on whether the used CRDTs are from the local library or from the distributed Selective Hearing library. The underlying mechanism should be transparent to the programmer. So, using a read operation to access the current value of a CRDT managed by Selective Hearing would not satisfy this goal. Instead, in object-oriented programming (OOP), it is more common to access the variables by their object reference. The use of object references imposes other complexities to the code. If, for example, the bind operation used a different object than the object referenced by the programmer, two diverging versions of a CRDT would be used.

**Algorithm 1:** Example CRDT in the Selective Hearing component

```
attribute: variable identifier vId
attribute: local state-based CRDT crdt
attribute: node-level monotonic counter clock
attribute: Selective Hearing instance sh
operation init is
    input     : unique name
    input     : identifier of this node nId
    input     : Selective Hearing instance sh
    vId.init(ExampleCRDT type, name);
    crdt.init(vId, nId);
    clock.init(nId);
    sh = sh;
end
operation merge is
    input     : updated instance other of crdt
    crdt.merge(other);
    clock.update();
    sh.bind(vId, this);
end
operation query is
    output    : data managed by the crdt
    return crdt.query();
end
operation update is
    input     : data to update the state of the crdt
    crdt.update(data);
    clock.update();
    sh.bind(vId, this);
end
operation join is
    input     : received ExampleCRDT other
    if clock does not contain other.clock then
        clock.update(other.clock);
        crdt.merge(other.crdt);
    end
end
```

**Algorithm 2:** Creation of an example CRDT in the Selective Hearing CRDT factory

```
attribute: node identifier nId
attribute: Selective Hearing instance sh
operation createExampleCRDT is
    input     : unique name name of the new CRDT
    output    : new Selective Hearing crdt wrapping an example CRDT
    crdt.init(name, nId, sh);
    return sh.declare(crdt.vId, crdt);
end
```

As more attention has to be paid to the object references, the original specification is not directly transferable to this project.

The following sections contain converted specifications in OOP pseudocode. The pseudocode algorithm blocks begin with a definition of the variables relevant outside this operation. The lines beginning with 'input' define parameters that are passed into the operation when it is called. The lines beginning with 'output' define the return values of the operation. No 'output' line indicates an operation without return value. For brevity, additional return statements are omitted from the operation. The lines beginning with 'attribute' define used attributes of the Selective Hearing object in which the operation has been called.

### Declare Operation

The declare operation, as specified by [35], is not complicated. Algorithm 3 shows the local part of the original declare operation. It sends the new identifier and does not change the node state.

**Algorithm 3:** Declare operation (as specified by [35])

```
input  : type information t
output: new variable identifier i
u = unique();
i = (u, t);
broadcast(declare, i);
```

The declare operation has been adapted in this framework. The following enumeration explains the details of the deviations and their rationales.

1. The provision of an existing identifier, that has been created from the calling party, has been allowed. This is a minor update but is required to allow the programmer to name the used CRDTs.

2. Instead of returning the new identifier, the object reference for the declared variable used inside the Selective Hearing object is returned. This is required to ensure the programmer uses the same object updated by Selective Hearing in the background. The communication layer does not guarantee any message order. So, a node might first receive a bind message containing an identifier before the variable is declared locally. In this scenario, an existing object would already be maintained in the known values set attribute of Selective Hearing. This object needs to be referenced from outside Selective Hearing.

3. Because the CRDT objects are initialized in a factory, these initial objects need to be provided to the operation. If a node has not yet received an object for this identifier, the provided initial object will be used and saved in the known values set attribute of the Selective Hearing object.

**Algorithm 4:** Declare operation (as implemented in this framework)

**attribute:** known values set $\delta_v$
**input** : variable identifier i
**input** : variable value object reference v
**output** : v = $\delta_v$.get(i)
**if** *not* $\delta_v$.containsKey(i) **then**
  $\delta_v$.put(i, v);
**end**
broadcast(declare, i);

Algorithm 4 shows the declare operation as implemented in this framework.

The declare operation receiving a new identifier has been implemented as specified. The identifier is added to the known variables set. In the current implementation, the known variables set might not be required at all. So, broadcasting and receiving new identifiers could have been removed.

### Read Operation

The read operation was not implemented in this project. It is not required in the current implementation. CRDTs maintained by Selective Hearing are accessed using the references of the concrete objects in the attributes of Selective Hearing. The programmer using the framework does not have to deal with specific CRDT implementations. Instead, it is possible to use whatever query and update operations are defined in the interface of the specified CRDT.

An operation similar to the read operation would be a good addition to the framework to prevent anti-patterns such as active or busy waiting [7]. If, for example, a piece of code should be executed once a CRDT reaches a specific value, it would be tempting to write a loop checking this CRDT and sleeping for a second if the value is not as awaited before checking again. A better method

is to register a function for later execution, combined with a predicate that is checked when the CRDT is updated.

**Algorithm 5:** Read operation (as specified by [35])

```
attribute: known variables set σ
attribute: known values set δ_v
attribute: interest set δ_I
input    : variable identifier i
input    : predicate operation p: v ↦ boolean
input    : continuation operation c: v ↦ void
if δ_v.containsKey(i) then
    v = δ_v.get(i);
    if p(v) then
        c(v);
        return;
    end
end
if δ_I.containsKey(i) then
    s = δ_I.get(i)
else s = ∅;
s.add(p, c);
δ_I.put(i, s);
σ.add(i);
```

Algorithm 5 shows the read operation as specified by [35]. The Selective Hearing attributes from the class implemented in this framework have been added to demonstrate that this operation could have been added without complications. However, to prevent requiring the programmer to know the library from which a CRDT is created, such an operation should not be added to Selective Hearing. Instead, a read operation would have to be added in the basic CRDT interface. In this design, the interest set would become a part of the CRDT attributes.

### Bind Operation

The two parts of the bind operation have both been adapted to fit the OOP world.

In [35], the local part of the operation is stateless. It just sends the broadcast message that binds a new value to an identifier in the receiving nodes. In OOP, more attention has to be paid to the object references. So, similarly to the declare operation, the local bind operation has been augmented to store the provided CRDT if none is available for this identifier. The line beginning the relevant block has been marked as added in Algorithm 6. This addition is not

strictly necessary, if the identifier is always declared locally before binding a value. However, to cope with programming errors, this addition has been left in the operation.

**Algorithm 6:** Bind operation (as implemented in this framework)

```
attribute: known values set δ_v
input    : variable identifier i
input    : new value v
added if not δ_v.containsKey(i) then
    δ_v.put(i, v);
end
broadcast(bind, i, v);
```

The receiving part of the bind operation always affected the Selective Hearing state. In the specification, this operation checked every pending predicate and continuation tuple in the interest set. The check of the interest set was omitted in this framework. Algorithm 7 shows the full bind operation, where the line beginning the omitted block has been marked as removed. If an operation similar to read had been implemented in a CRDT, this block would have to be called by the CRDT whenever it is updated.

**Algorithm 7:** Receiving bind message operation (as specified by [35], implementation deviations have been highlighted)

```
attribute: known values set δ_v
attribute: known variables set σ
attribute: interest set δ_I
input    : variable identifier i
input    : new value v
if δ_v.containsKey(i) then
    δ_v.get(i).join(v);
else δ_v.put(i,v);
σ.add(i);
removed if δ_I.containsKey(i) then
    v_n = δ_v.get(i);
    s = δ_I.get(i);
    s_sat = s.filter(p ↦ p(v_n));
    s.removeAll(s_sat);
    δ_I.put(i, s);
    foreach c ∈ s_sat do
        c(v_n);
    end
end
```

## 3.5 Conclusion

In this chapter, the framework architecture has been presented. The framework is separated into different packages. All components are loosely coupled. The architecture follows OOP best practices. Design patterns, such as the observer or abstract factory pattern, have been utilized throughout the architecture. Other design principles such as dependency injection [12], the dependency inversion principle [30, 29], or the interface segregation principle [31] have been respected. The CRDT library provides interfaces that application developers can consistently rely on, no matter which implementation is used. A blueprint for a wrapped CRDT, as implemented in the Selective Hearing component, has been provided. The Selective Hearing algorithms required some adaption for the OOP world. These deviations from the functional paradigm were presented. The following chapter highlights notable topics from the framework implementation.

# 4 Implementation

This chapter gives insights into the implementation of the framework architecture, which has been described in the previous chapter. Important decisions are rationalized, and alternatives along with their advantages and disadvantages, are provided.

## 4.1 Generic state-based CRDTs

Some CRDTs allow the application to maintain various types of data. In this project these are the register CRDT, all sets, and the top-$k$ list. The algorithms for each of these CRDTs may be used with any data type. The exception is the top-$k$ list, which was also limited to only allow classes that can be put into order (through the comparable Java interface). These CRDTs have been implemented using Java generics [5, 6] to achieve the algorithms' property in the framework.

CRDTs could be stored inside a CRDT as a consequence of the generic implementation. In [44], such CRDTs are called 'embedded CRDTs.' Hovewer, embedded CRDTs have not been tested in this framework. Generic CRDTs have only been tested using integers. Additionally, CRDTs, with special semantics supporting embedded CRDTs, have not been considered in this project.

Type variables in Java do not exist at run-time [5, 6]. All generic type information is erased during the compilation. This erasure process prevents the usage of generic types in casts. An 'unchecked warning' is given if generic types are used in casts. [5] states that "if your entire application has been compiled without unchecked warnings [...], it is type safe."

State-based CRDTs' implementations need to cast a received CRDT into the same class before a merge is possible. This cast is required because different CRDT implementations may hold their state in incompatible ways. The merge operation requires compatible states. Additionally, it accesses the private attributes maintaining the state. When two incompatible CRDTs are merged, a specific exception is thrown.

The unchecked warnings have been accepted in this framework. They are a consequence of the interface in the CRDT library. In the interface, the merge operation accepts any state-based CRDT. Its design allows the interface to be used in the variable values of Selective Hearing and with any wrapped CRDT.

## 4.2 Serializable CRDTs

An essential part of a middleware is the marshaling and unmarshaling of the provided data types [53]. The serializable interface has been used in this Selective Hearing implementation to pass CRDTs between nodes. This choice reduced the required development overhead.

The disadvantage of serialization is that only CRDTs from compatible framework versions can be exchanged. Different versions of the framework could cause errors. It is not possible to pair the framework with a system based on a different environment. However, this limitation is not an issue when evaluating a proof-of-concept. The requirements did not specify the need to exchange CRDTs with different systems.

A substantial advantage of serialization is that generic CRDTs are supported out-of-the-box. So, no special handling for generic CRDTs had to be developed. A bounded wildcard [5, 55] has been used in the generic CRDTs. This bound limits the classes maintained in a generic CRDT to any subclass of the serializable interface.

The fast-serialization (FST) package [45] has been used in this implementation. This dependency provides a drop-in replacement for the built-in Java Development Kit (JDK) serialization. Apart from the faster serialization, FST offers the ability to use JSON, which could be used for interoperability with other systems in the future. However, this ability should not raise false hopes for the required efforts for interoperability.

## 4.3 Identifiers

Identifiers were used throughout the project. Nodes, as well as Plumtree messages, and CRDTs need to be identifiable in Selective Hearing. CRDTs were designed to know their identity. Multiple implementations of the identifier interface exist. Some are general, like a universally unique identifier (UUID) identifier or a string identifier similar to a variable name. Additionally, custom identifier implementations have been developed for special needs. Specialized identifiers in this framework are the Plumtree message identifier or a Selective Hearing variable identifier.

### 4.3.1 Node Identifiers

A node's name can be supplied as an argument when executing the application. A string identifier is created from the name if it is present. Otherwise, a new UUID is used. In both cases, the node's identifier is a global constant accessible through a static inner class of the identifier interface.

Nodes need to be identified consistently in the HyParView group. The node identifiers are communicated in messages as strings. The received identifier string is treated in the same way

as the local node identifier. So, all nodes in the group are currently required to use the same identifier class. This approach is not optimal. Consistently using the name identifiers would be preferable. Of course, a string representation of a UUID could be used to create a random name.

The Plumtree implementation receives most of the node identifiers it needs from the peer gossip membership service. The nodes it needs to access are HyParView implementations of the peer interface. Plumtree also uses the global node identifier to identify itself.

The global node identifier is also used in some CRDTs. Affected CRDTs are named CRDT, where all updates or elements contain the name of the replica to which they have been submitted. The positive-negative counter CRDT is an example of a named CRDT. The CRDTs might not be required to use the node identifiers. However, this usage has not caused any issues.

### 4.3.2 CRDT Identifiers

Requirement 3 of this project is to prevent merging CRDTs with different origins. The origin has been interpreted as being an identifier. So, each CRDT is instantiated using an identifier as a global name across all nodes. CRDTs from the CRDT library reject merging CRDTs created with a different identifier by throwing an exception.

CRDT identifiers have an effect on the CRDT factory design. The factory methods need to construct the CRDTs with an identifier. Two distinct ways to provide names are supported in the factory interface: The first takes an identifier as input, and the second takes a name string. A provided identifier is used verbatim in the local factory. If a string is provided to the factory method, it is turned into a name identifier before using it in the CRDT.

The addition of names to the CRDT instances was an early decision in the project. CRDT specifications do not require CRDT names. The solution tries to solve the naming issue in a layer that should not be responsible for solving it. A list instance in Java is not required to know the name the programmer has given it. Instead, the list's name is only relevant outside of the list data structure. Similarly, the Selective Hearing algorithms use identifiers to maintain CRDTs.

Using an identifier has a minimal benefit for casting the interface to the generic type. As described previously, the generic type cannot be guaranteed at run-time. A CRDT identifier could serve as a weak layer to reduce issues with conflicting types in generic CRDTs. However, as identifiers can be used in incompatible CRDTs, this is no guarantee.

Providing identifiers in CRDTs can be considered as technical debt in the framework.

### 4.3.3 Variable Identifiers in Selective Hearing

In Selective Hearing, variable identifiers consist of the type and a unique name for the variable. An instance of the class has been utilized as type part for the variable identifiers. The variable identifiers are used in two distinct ways. First, they are submitted to the local CRDT factory. Second, they are used in the Selective Hearing state to match received bind messages with existing CRDT instances.

The Selective Hearing CRDT factory uses the string representation of the supplied identifier to create a variable identifier instance. A provided name is used verbatim.

### 4.3.4 Message Identifiers in Plumtree

Plumtree needs to identify messages to check what message a node has already received and which needs to be requested in the lazy push manner. The message identifier is a hash created from the original sender of a message, the message name, and the message content. The advantage of this construction is that the message identifier can be consistently generated from a message without requiring that the identifier is transferred.

## 4.4 HyParView using Netty

Netty is a network communication framework. Its network primitives were used to develop the HyParView protocol. Netty follows an asynchronous event-based programming paradigm. This paradigm matches the protocol specification in [26]. However, the asynchronous approach also causes some difficulties, as many updates may happen anytime and parallel to others.

### 4.4.1 Message Passing

Lower communication layers should not need to care what type of data they are sending. So, the communication layer interfaces do not support any special objects. Instead, strings are used in all interfaces of this framework to pass message data through the layers.

Strings are directly supported by Netty using provided encoder and decoder classes. This existing support reduces the development effort required to implement the communication layer. An additional advantage is that using strings achieves consistency when combining the different layers.

However, a string is not the most efficient way to transport all different kinds of data. While some data structures, such as JSON, have natural string representations, other complex data structures

used in the higher layers should not need to be encoded into a string. In this framework, the FST package serializes objects into byte arrays. To be able to send these serialized objects, the byte arrays first had to be encoded into strings. This increases the message size and reduces performance. Ironically, Netty converts everything into byte arrays before sending it. Using byte arrays directly to pass parameters between the different layers would reduce the amount of data sent across the wires and remove duplicate encoding efforts between the layers.

Other alternative methods for passing parameters between the layers have been considered. Data formats such as MessagePack[1] or protocol buffers[2] have not been used to increase the implementation simplicity.

Another option would be to allow a HyParView user to supply a custom encoder for the Netty channel pipeline that processes each incoming and outgoing message. Such an option would allow more fine-grained handling of received messages. However, it would expose the HyParView dependency on Netty in the technology-agnostic interface.

### 4.4.2 HyParView Peer

The gossip membership protocol HyParView is responsible for maintaining the peers known to the group. The HyParView package implements the peer interface from the broadcast package. This HyParView peer initiates the TCP connection to a node. In Netty, such a connection creates a distinct channel between the nodes. This channel is used to send and receive data. The HyParView peer manages the created channel. The channel is also tracked to detect changes by the peer. If the connection is lost, it notifies the HyParView protocol, which then notifies the registered neighborhood observers.

## 4.5 Conclusion

The implementation of this CRDT framework brought a variety of programming paradigms together. This chapter explored far-reaching decisions made during development. The state-based CRDTs have generic implementations. Generic CRDTs allow for flexible usage of data types. However, it reduces the type-safety of the CRDT library. The CRDTs of this framework are serializable. The FST package FST is used for fast serialization. Various components must be uniquely identifiable. This identification requirement is achieved using an identifier interface with reasonable different implementations. The gossip membership protocol HyParView has been

[1] MessagePack website, https://msgpack.org/, accessed 2021-07-10
[2] Protocol Buffers website, https://developers.google.com/protocol-buffers, accessed 2021-07-10

implemented, relying on the Netty framework for network communication. The implemented framework is evaluated in the following chapter.

# 5 Evaluation

In this chapter, the implemented framework is evaluated. The setup of the evaluation environment is explained in Section 5.1. The purpose and findings per metric are presented in Section 5.2. The overall results of this evaluation and the framework are discussed in Section 5.3.

## 5.1 Environment Setup

In the evaluation, a dummy application that uses the framework is executed in a Mininet environment. The execution is performed with different group sizes and with different packet loss rate configurations. During the execution, logs are created, which are used to compute metrics.

Mininet [25] is a container-based emulation allowing the creation of virtual networks on a single computer. It uses the virtualization mechanisms of the Linux kernel to simulate multiple nodes. Nodes are called 'hosts' in Mininet; therefore, this chapter uses this terminology. Each host has an individual network interface and can execute the applications that are available on the computer. The virtual network can be constructed using a Python script. This script can also be used to execute software on the virtual hosts. Using Mininet with a Python script enables reproducible evaluation, which would not be as easily possible in a hardware setup.[16] The downside of this approach is that every host is executed on the same hardware. This setup can limit the computational resources available to each host.

The Mininet network topology that has been set up for the evaluation of this project contains a single switch, which is connected to each host in the system. This topology simulates a broadcast domain and is shown in Figure 5.1. The number of hosts connected to the switch has been scaled from 3–23. Three has been chosen as the lower bound because this is the current number of Loomos in the CaDS Loomo fleet. As Loomos are expensive, a Loomo fleet will probably not be large. A typical tourist group size, using other Segway vehicles in Hamburg, is around 15[1]. The evaluation of the framework is executed with up to 23 hosts. This also covers a scenario with 15 Loomos and some additional devices, such as smartphones. In addition to different

[1]Based on personal observation, and a short web search.

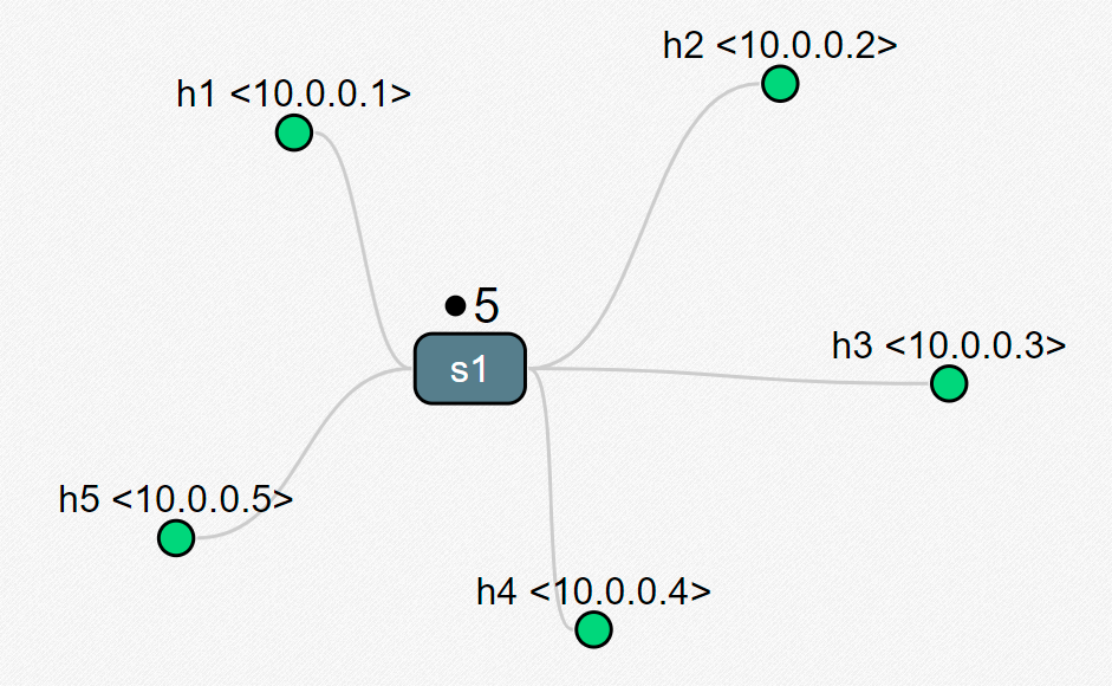


Figure 5.1: Example topology of the Mininet network used in the evaluation. Five hosts are connected to a single switch. Rendered using [51].

group sizes, different link packet loss rates have been configured in Mininet. Because the switch is in the center of the topology, each message sent between two hosts passes two links. These transmissions can cause the message's packets to be lost twice. In addition to 0 % packet loss, the losses have been scaled exponentially from 0.2 % to 25.6 %.

Each host executes an evaluation application that has been built using the framework. The evaluation application is simple. It uses an injected CRDT factory to initialize a counter CRDT. In the evaluation, the Selective Hearing CRDT factory instance is used. The application is configured with a global maximum for the counter and a maximum of increments that may be performed on the local host. In this dummy application, a host queries the counter for its current known value. If the global maximum has been reached, the application is finished. However, the host and the gossip protocols continue their execution on this host. If the global maximum has not been reached and the local maximum number of increments for this host has not been reached, the counter is incremented by 1. The application process then sleeps for a second before performing the next iteration. This happens in both cases, whether the local increment count has been reached or not. The local increment count has been set to 0 for all but one host in the evaluation. A single host in the evaluation is configured to increment the counter CRDT once.

The framework's Plumtree package has been expanded with detailed logging outputs. Each message that is sent or received in the Plumtree package is logged. Messages sent by a host

| parameter | value | unit |
|---|---|---|
| active view size | 4 | hosts |
| passive view size | 5 | hosts |
| active forward join random walk length | 3 | hops |
| passive forward join random walk length | 2 | hops |
| active shuffle list size | 3 | hosts |
| passive shuffle list size | 3 | hosts |
| shuffle TTL | 3 | hops |
| interval for new shuffle requests | 4 | seconds |

Table 5.1: Configuration of the HyParView protocol used during the evaluation.

are logged immediately before the method call, which sends a message to the specific neighbor. Similarly, before a host processes a received message in Plumtree, a line is written to the logs. As all broadcasts by Selective Hearing are sent through the Plumtree layer, the logs contain sent and received information for all Selective Hearing and Plumtree messages. HyParView messages have not been logged during this evaluation. Each line in the logging data includes the action (was the message sent or received), the host writing the log, the message's sending host, the message's receiving host, the message name[2], the message identifier, the hop count, and a timestamp. All log outputs are combined into a single CSV file per execution round. This file is parsed in a Python script that extracts the required values to calculate the metrics.

The application is executed ten times for every group size and loss combination. An average of these ten results is used in all metrics, except for the difference between sent and received messages. The reason it is not used in this difference is elaborated upon in section 5.2.1. Using the average to compute the metrics reduces the temporary effects of other processes on the runtime environment.

Each of these evaluation rounds takes 20 seconds and is executed in two phases. In the beginning, the membership protocol has 5 seconds to build a membership group. The subsequent 15 seconds are used to run the example application.

Table 5.1 shows the configuration used for HyParView during the evaluation. Other parts of the framework, such as Plumtree, have not been configured. However, the active view size of 4 is also the number of hosts used as peers by Plumtree.

The evaluation was developed on a laptop. However, this device was too insufficiently powered to run the dummy application on more than eight Mininet hosts. In order to support larger group sizes, the experiments have been executed on a CaDS group's server.

[2]The message name consists of the Plumtree and the Selective Hearing message type.

| Section | Metric | Parameter |
|---|---|---|
| 5.2.1 | sent − received messages | host number; packet loss rate |
| 5.2.2 | reliability | host number |
| 5.2.3 | RMR | packet loss rate |
| 5.2.4 | LDH | host number |

Table 5.2: Evaluations overview with their metrics and adapted parameters.

## 5.2 Analyzed Metrics

This section explores the purpose, origin, calculation, results, and observations per analyzed metric. An overview of the metrics can be found in Table 5.2. This table also contains the corresponding sections in which each metric is discussed.

### 5.2.1 Difference Between Sent and Received Messages

This metric aims to detect errors in the evaluation setup and verify that the evaluation environment produces reliable data. Reliable data, in this context, means data that can be used in further investigations to compute the other metrics correctly.

The first expectation for this metric is that the minimum number of lost messages is never negative. As elaborated upon in Section 5.1, each message is logged by the host when sent and received. A message is lost when it is logged on the sending node but not on the receiving node. No indication was found in the Mininet resources (e.g., linked from [24]) that Mininet duplicates messages. Based on this setup, it can be concluded that the simulated hosts should never print logs containing more received than sent messages. If this is verified, it is an indication that the evaluation environment produces reliable data. However, it does not indicate that there are no errors in the implementation. If more received than sent messages are observed in one execution, the evaluation setup does not produce reliable data.

The second expectation is that some messages are lost in scenarios where the packet loss rate exceeds 0 %.[3] Additionally, the amount of message loss is expected to increase with increasing loss rates. The number of nodes per group is not expected to affect the number of lost messages.

To calculate this metric, all lines in the logging output are counted grouped by their action value, which can either be sent or received. These two values are subtracted to receive the number of lost messages. The metric is calculated for each of the ten executions per combination. Each execution per combination yields a different number of lost messages. The lost messages minimum per combination of group size and packet loss rate has been used for this metric. Using an average

[3] Please note: lost messages do not necessarily affect the reliability value, as discussed in the next section.

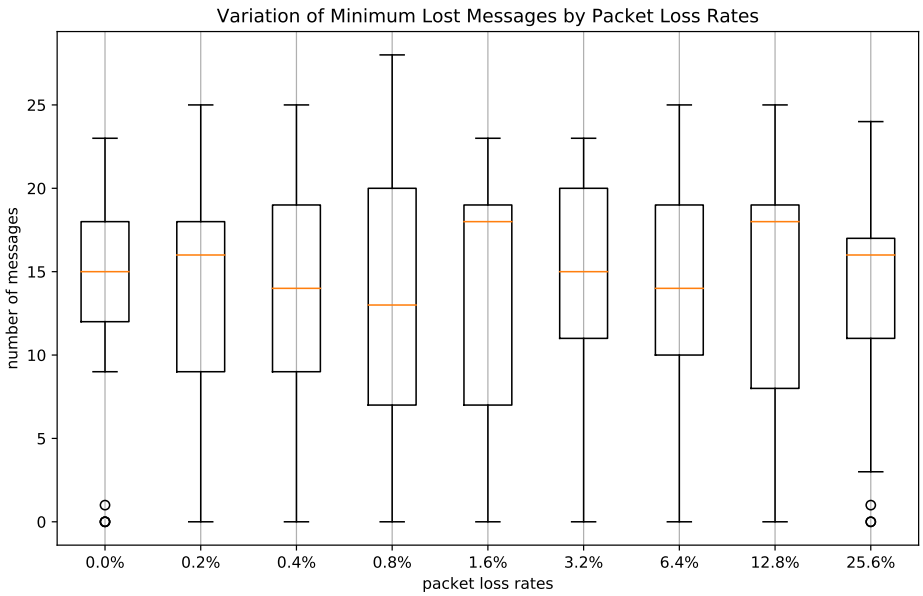


Figure 5.2: Box plot diagram showing the minimum number of lost messages by the used packet loss rates.

value of the ten executions could have hidden negative differences. The minimum value does not guarantee that every negative value can be observed. However, if negative differences appear, at least one negative difference is visible. Moreover, as the minimum value is used, a message loss of zero does not indicate 100 % reliability in Section 5.2.2; it just means that no message was lost in one of the ten executions.

Figure 5.2 shows box plots[4] of the minimum number of lost messages. One box plot is pictured for each packet loss rate used in this evaluation. The box plots are created from the values of the 21 evaluated group sizes. A line chart showing the individual group sizes can be found in the appendix in Figure A.2.

The lowest number shown in the box plots in Figure 5.2 is zero. This result meets the first expectation. Thus, it can be presumed that the evaluation environment did not create apparent mistakes in the evaluation data for the other metrics.

However, surprisingly contrary to the second expectation, the packet loss rates do not considerably affect the minimum message loss. This effect should have been observable in Figure 5.2 by higher boxes on higher loss rates. Instead, the boxes do not show a tendency for higher message losses. Even in the box plot for the packet loss rate of 25.6 %, one outlier is zero.

[4]To describe the box plots, the notation from [57] is loosely followed.

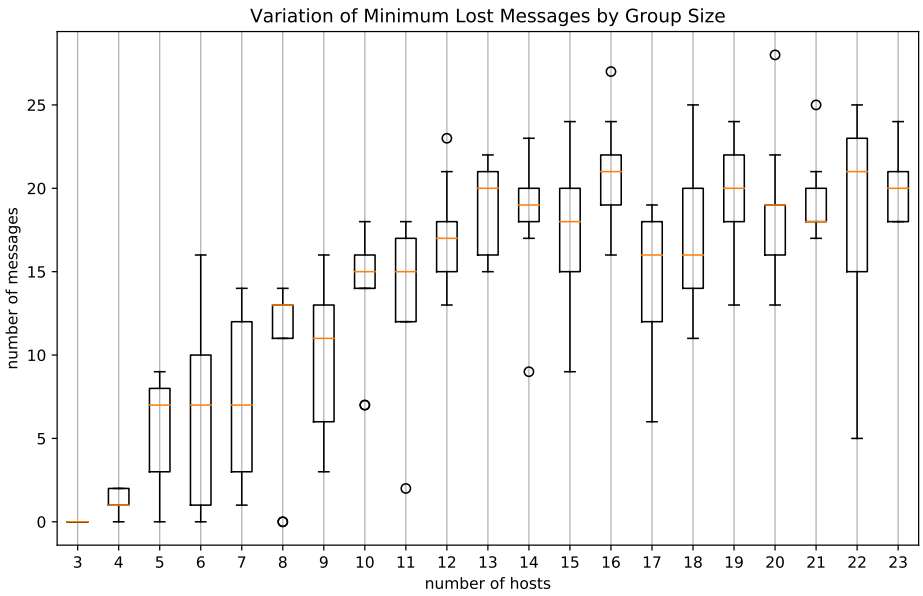


Figure 5.3: Box plot diagram showing the minimum number of lost messages by the used group size.

This value indicates that one execution round with a 25.6 % loss rate did not lose any message sent by the Plumtree protocol.

Figure 5.3 shows box plots of the minimum number of lost messages. One box plot is pictured for each group size used in this evaluation. The box plots are created from the values of the nine evaluated packet loss rates. A line chart showing the individual packet loss rates can be found in the appendix in Figure A.3.

Again, contrary to the second expectation, the group size shows an effect on the minimum number of lost messages. Figure 5.3 shows increasing numbers of lost messages with increased group sizes. However, the sizes of the boxes and their whiskers are inconsistent. The group size three is noticeable, as its minimum number of lost messages per packet loss rate is never above zero. This outcome explains the outlier in Figure 5.2 for a packet loss rate of 25.6 %. The small group sizes with up to five nodes show upper extremes of minimum lost messages below ten messages. Beginning with a group size of nine, the minimum number of lost messages is never zero. Starting with around 13 nodes in a group, the minimum number of lost messages seems to have plateaued.

An explanation for the increasing minimum number of lost messages is that packet loss rates affect every message. The Plumtree protocol sends fewer messages than the HyParView protocol. So, it is more likely that HyParView messages are lost. Lost messages in the membership layer

can lead to disconnected nodes. A disconnected node does not send messages, thus further decreasing the number of messages sent by the Plumtree protocol. This disconnection can be observed by the fact that in multiple execution rounds, the bind message, which changes the counter value, was never logged. This observation indicates that the host, which is configured to change the counter, is not connected to any other host in the group and thus does not send the message.

This metric helped fix mistakes in the early development of the evaluation environment. All messages were logged when they were received at one point in the development, but only some sent messages were logged. Another issue observed was the runtime environment. Running the evaluation in a virtual machine on a decent but not high-end laptop resulted in a loss of a single line at some points. This loss did not occur when running the experiments on a bare-metal server.

### 5.2.2 Reliability

This metric investigates the question of whether this framework can work reliably in a small-sized node group. All messages from the Selective Hearing component are sent through the Plumtree communication layer. The reliability can thus be observed in the Plumtree layer. In [27], the reliability of the Plumtree protocol has been evaluated. The same reliability definition is used in this evaluation, stating that "reliability is defined as the percentage of active nodes that deliver a gossip broadcast" by [27, p. 4]. In Section 5.3, the comparability between the results of [27] and this thesis is discussed.

The reliability definition, quoted in the previous paragraph, can be used as a distinction from the lost messages metric (5.2.1). The latter metric shows the reliability of the underlying virtual network. As this virtual network has been intentionally configured with packet loss rates, message loss is expected. The metric, as defined for the Plumtree protocol, shows the reliability of the communication layer. Whether packets are lost is not relevant for the reliability metric. Instead, the reliability metric investigates how well the algorithms mitigate packet loss to deliver a broadcast to every active host.

The expectation is that the packet loss rates only minimally affect the reliability. After all, the purpose of the Plumtree protocol's lazy push phase is to achieve high reliability. It is suspected that a minimal number of hosts is required in a group. So, the reliability is expected to begin at lower rates and increase, starting with a certain group size.

To calculate the reliability, the number of hosts that received a broadcast message were counted from the log file. The evaluation Python script calculated the reliability per gossip message in each execution round. Control messages from the Plumtree protocol, such as the prune and graft

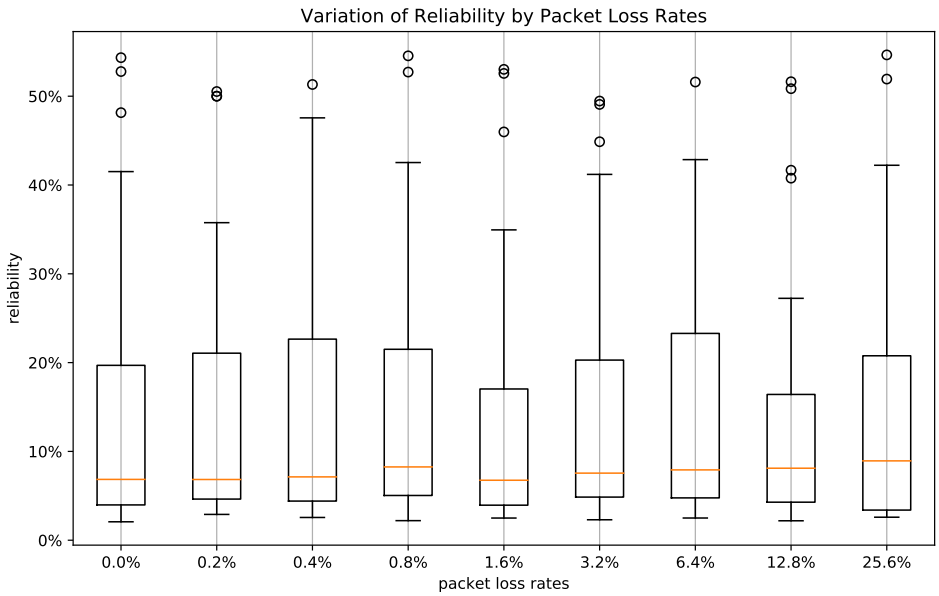


Figure 5.4: Box plot diagram showing the reliability by the used packet loss rates.

messages, were ignored in this calculation because they are sent directly between two hosts and not as a broadcast. The collection of percentages was aggregated into an average percentage.

Figure 5.4 shows box plots of the reliability. One box plot for each packet loss rate used in this evaluation is pictured. The box plots are created from the values of the 21 evaluated group sizes. A line chart showing the individual group sizes can be found in the appendix in Figure A.4.

The box plots in Figure 5.4 clearly show unexpectedly low reliability results. Only a few outliers and no upper extreme of the box plots reach above 50 % reliability. Even the best scores in this diagram are not nearly enough for productive use. In every box plot, the median is below 10 % reliability.

The box plots in Figure 5.4 look similar for each of the packet loss rates. This outcome indicates that the loss rates only had minimal impact on the reliability. As the box plots were rendered from the results that different group sizes achieved for each loss rate, they meet the expectation that a certain number of hosts is required for higher reliability. The worse than expected results could hint that the number of required hosts is above the maximum group size used in this evaluation. However, the line chart of this metric, as shown in Figure A.4, and the following diagram tell a different story.

Figure 5.5 shows box plots of the reliability. One box plot for each group size used in this evaluation is pictured. The box plots are created from the values of the nine evaluated packet

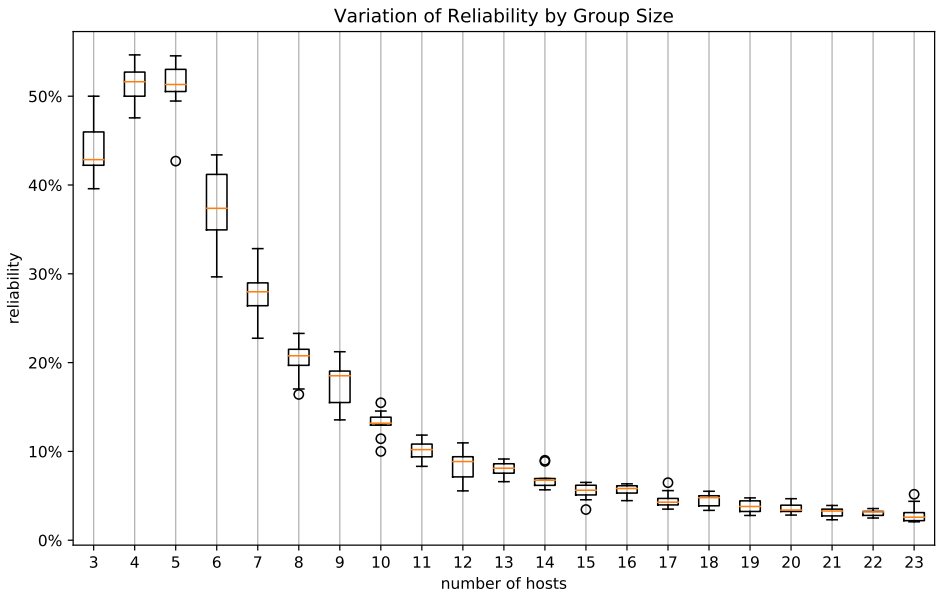


Figure 5.5: Box plot diagram showing the reliability by the used group sizes.

loss rates. A line chart showing the individual packet loss rates can be found in the appendix in Figure A.5.

The box plots in Figure 5.5 show that the reliability developed contrary to the expectation stated above. Instead of increasing with the number of hosts per group, the reliability steeply decreases between the group sizes 6–11. The reliability continues to decrease from 12–23 hosts per group, though less steeply. The highest reliability results were reached with a group size of 4 and 5 hosts. Apart from the groups with three, six and seven nodes, the upper and lower extremes of the box plots were close together. As expected, the values achieved with different loss rates shown here are another indication that the loss rates have only a minimal impact on the reliability.

### 5.2.3 Relative Message Redundancy

With the Relative Message Redundancy (RMR) metric, the expected gossip message overhead is evaluated with different node numbers in the group and packet loss rates. The RMR metric is defined by [27] as:

$$\left(\frac{m}{n-1}\right) - 1 \tag{5.1}$$

In this equation, $m$ is the total number of gossip messages during a broadcast. $n$ is the number of nodes in the group that have received the broadcast.

An RMR value of zero indicates that exactly one message per host was sent. This is the optimal value. Higher RMR values indicate that redundant messages have been sent to reach every node. A negative value would indicate that more nodes have received the broadcast than messages have been sent. [27] mentions that "this metric is only applicable when at least 2 nodes receive the message." Moreover, it states that a low RMR can be achieved with low reliability, which was the case in this evaluation.

$n$ includes the host sending a broadcast message in the first place. This is not explicitly defined in [27], but it needs to be the case to make zero the optimal value. In a group of four nodes, $a, b, c, d$, if $a$ sends a broadcast message to $b, c$ and $b$ forwards it to $d$, then three messages are sent. First, if $n$ does not include the sending node, the equation 5.1 is computed with $m = 3, n = 3$: $\left(\frac{3}{3-1}\right) - 1 = -0.5$. This can be compared with using $m = 3, n = 4$: $\left(\frac{3}{4-1}\right) - 1 = 0$.

The first expectation is that the RMR results increase for higher loss rates. When more messages are lost due to the packet losses, these messages need to be sent again by the Plumtree algorithms. Each resent message increments the $m$ in Equation 5.1, while $n$ is only incremented for received messages.

The second expectation is that a higher RMR is achieved for bigger group sizes. This expectation combines the observations from the previous metrics. In Figure 5.3, it is apparent that big groups lost many messages, leading to a high $m$ in Equation 5.1. Figure 5.5 provides evidence of low reliability in groups with 11–23 hosts, leading to a low $n$ in Equation 5.1.

It is also expected that only positive RMR values are reached. As explained above, a negative RMR would indicate an error in the evaluation environment or the calculation, similar to a negative number of lost messages.

To calculate the RMR, the messages sent by Selective Hearing have been observed. Control messages by the Plumtree protocol were not considered. The $m$ was built from the log, where the action indicates the sending of a message. Each host that has either sent or received a message was counted once in $n$ of Equation 5.1. The computation of the RMR value only continued if $n \geq 2$. If a message was not received by at least two hosts, the message was ignored in this metric.[5]

Figure 5.6 shows box plots of the RMR. One box plot for each packet loss rate used in this evaluation is pictured. The box plots are created from the values of the 21 evaluated group sizes. A line chart showing the individual group sizes can be found in the appendix in Figure A.6.

[5] A negative impact for a message that was not received by two hosts is reflected in the reliability metric.

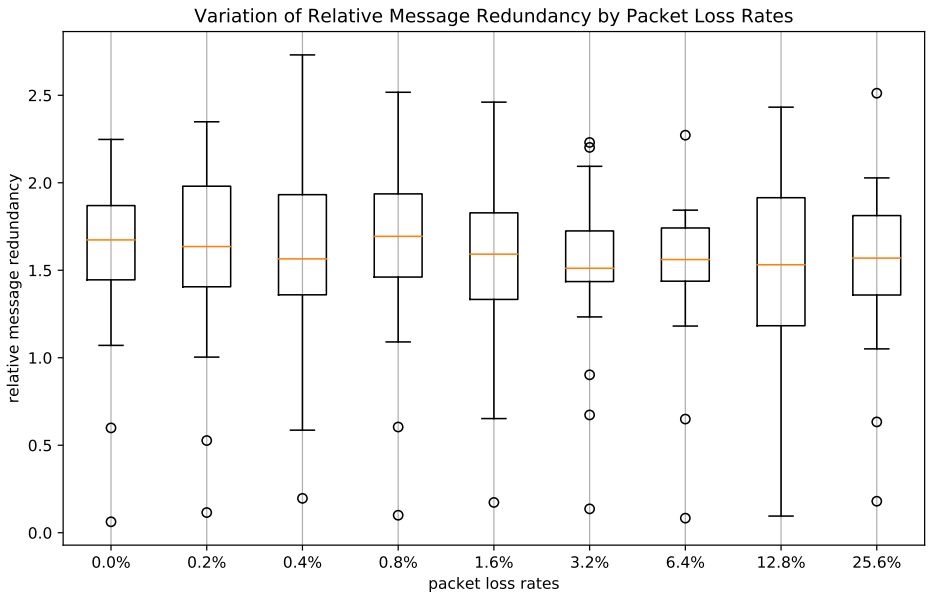


Figure 5.6: Box plot diagram showing the RMR by the used packet loss rates.

Just like the previous metrics, Figure 5.6 shows that the RMR results are only minimally affected by the packet loss rates. This outcome is contrary to the first expectation. It could be explained by the condition that only broadcasts where two or more hosts received a message ($n \geq 2$) were considered in the calculation. This condition could remove broadcasts where many messages have been sent but were never received. As described previously, if many HyParView messages are lost, the overlay can be disconnected. In such a scenario, many broadcast messages would never be sent at all. According to the third expectation, the RMR is never negative.

Figure 5.7 shows box plots of the RMR. One box plot for each group size used in this evaluation is pictured. The box plots are created from the values of the nine evaluated packet loss rates. A line chart showing the individual packet loss rates can be found in the appendix in Figure A.7.

The results depicted in Figure 5.7 show a steep growth of the RMR in groups with 3–6 hosts. The value flattens in an RMR range from 1.5–2.5. This is according to the second expectation for the results of the RMR metric. However, the higher and less intended RMRs are still not that big. Given that "in pure gossip approaches, RMR is closely related with the protocol fanout, as it tends to $fanout - 1$"[27, p. 4], in this environment, the fanout is initialized from the active view size, which is 4. Two restrictions of RMR can explain this result. First, it is only applicable when at least two hosts have received the broadcast and thus $n \geq 2$. Broadcasts where $n < 2$ have not been considered. Second, as mentioned before, the RMR metric depends on the reliability results. The RMR values are not very informative because the reliability achieved by the framework is

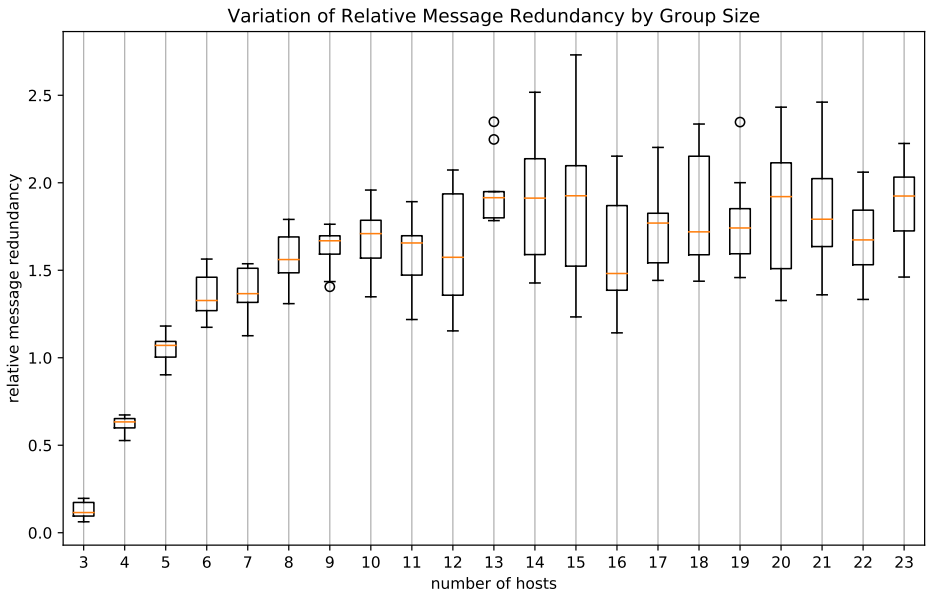


Figure 5.7: Box plot diagram showing the RMR by the used group sizes.

low. Additionally, due to the difference in reliability, the RMR result cannot be compared to the result of the original Plumtree protocol.

### 5.2.4 Last Delivery Hop

The Last Delivery Hop (LDH) metric allows the investigation of a Plumtree property.[6] Plumtree builds a spanning tree covering all hosts in the group. Each host has a limited number of other hosts as children in the spanning tree because the children per host are limited by the number of hosts in the membership service's active view. In this evaluation, it is limited to 4, as explained in section 5.1. These experiments scale the number of hosts in the group while the active view size stays constant. Due to the constant active view size of 4, the spanning tree cannot grow in breadth; instead, it needs to grow in depth. This property should be observable with higher LDH values in this framework's Plumtree implementation.

The logging output always contains the current hop value for the logged message. The maximum of observed hop counts in multiple executions of 0 % message loss has been used.

[6] It is not attempted to mathematically formulate and proof this property. However, it might be linkable to the height of a B-Tree as in [9].

With an active view size of 4, the LDH might be the same for groups with 3–21 hosts. This property should inevitably become observable, beginning with 22 hosts per group, even with a balanced spanning tree, provided the communication protocols have high reliability. Reliability is not high in this framework, as laid out in section 5.2.2. So, it is not surprising that the LDH does not increase in the result of this thesis. This outcome is linked to the high message loss (Section 5.2.1) and low reliability (Section 5.2.2).

The LDH showed a constant maximum value of 2 in every group size. This outcome indicates that the Plumtree protocol prunes links, even in small groups. The Plumtree mechanisms reduce message duplication, even when only a few hosts participate.

## 5.3 Discussion

The evaluation of this framework yielded less appreciable outcomes than desired. In this section, potential causes for these outcomes are discussed, as well as their effect on the success of this project.

The framework achieved such low reliability in the evaluation that it cannot be considered usable in small groups. If implementation errors caused the low reliability, another implementation of the framework, or parts of it, could achieve better results. If a scaled-down group requires differently tweaked parameters, a different configuration could achieve higher reliability. The design of the used protocols could require a certain higher number of nodes. The framework might never be usable in small groups if such a requirement is the case. The framework consists of multiple parts. It may be only a single component that causes the low reliability. Due to the architecture design, the components are lowly coupled. A faulty component could be exchanged with a fixed component. Hence, the whole framework need not to be considered unusable in small groups.

The low reliability could be linked to discovered anomalies in the HyParView implementation. In some log files of the executions, a bind message sending action was never tracked. No host can receive a message that has never been sent. Plumtree does not send a message when it has empty eager or lazy push peers sets. These sets are empty if HyParView does not have a peer in its active view.

Such a scenario can occur if messages from the HyParView join process are lost. This process offers a lot of opportunities for a message to be lost. The process does not start if the new node's initial join message is lost. The current HyParView implementation does not retry to establish a connection if the first attempt fails. However, if the join node receives the initial message, its direct answer message to the new node can be lost. Losing this neighbor request is not necessarily a problem; however, it reduces the number of early group connections for the new node. In addition to the neighbor message, the join node sends forward join messages to its

existing active view nodes. Each of these forward join messages is passed on a few nodes before the new node is added to a passive or active view. The forward message could get lost each time. In some circumstances, this could lead to a situation in which a new host is never connected to the group.

A new node could succeed in establishing a connection with the group, even if a few messages from the join process are lost. This established connection could vanish. Firstly, if packets do not arrive, a connection can become idle and may be closed. However, idle connections are unlikely in the short time scope of this evaluation. Secondly, if a node from the active view disconnects, the node tries to initiate a connection with a node from the passive view. This attempt could fail and leave the node disconnected. A node might disconnect if it receives a new neighbor request with high priority and already has a full active view. Thirdly, when a node from the active view disconnects, the node can only establish a new connection if its passive view is not empty. Unfortunately, a passive view could be empty if a node never receives messages the protocol uses to fill a passive view. These are the forward join, disconnect, and shuffle messages.

The HyParView issues might result from potential errors in the HyParView implementation. The HyParView is implemented in an asynchronous event-driven style, where most operations are executed in parallel and at some time in the future. Due to time constraints in the project, a dependable protocol testing strategy has not been implemented. In earlier experiments, the expected basic operation of the protocol was observed. However, this does not eliminate the potential for problems with increased group sizes.

Errors in the configuration of the protocols are more likely to cause the HyParView issues. The protocol parameters in this evaluation have been chosen, influenced by the experiments in [27]. However, these experiments were conducted in a different network and group context. In this framework evaluation, many fewer nodes have been used than in the experiments in [27, 35, 26]. It has been observed that protocol parameters might not be clear when scaling experiments down.[16] In the evaluation, it is not easy to distinguish between configuration errors and unfitness.

In [27], protocol parameters from previous simulation experiments with a similar setup were chosen. Relying on previous experiences is not possible in this project because this thesis is the first attempt at using these gossip protocols in the CaDS group. So, instead of experimenting with different network properties (e.g., the link packet loss rate), it would have been wiser to begin with an investigation to determine suitable protocol parameters for this evaluation's group sizes. As the Plumtree implementation was not configurable, a good starting place would be the HyParView parameters: active and passive view size, active and passive forward join walk length, shuffle TTL, number of active and passive peers in a shuffle message. As these parameters

define the number of hosts for different purposes, they are most likely affected by the number of hosts in the group.[7]

The selected protocols have been designed for large-scale networks, as described in Section 2.3 and indicated by the number of hosts used in their experiments. This could also lead the algorithm design to require a minimum number of hosts in a network to work as intended. Another indication supporting this idea is that 30 hosts have been used for the HyParView passive view in [26]. This passive view size would not have been possible with the group sizes of this thesis' evaluation. In addition to the passive view size, the random walk performed by HyParView in the forward join or shuffle operations may require larger groups. The random walk is less diverse in small groups than required to achieve failure compensation.

As noted previously in this chapter, the LDH results indicate that the Plumtree algorithms already performed their operations with small groups. Prune, and graft messages have been observed in the log files. So, even with only a few hosts, Plumtree reduced redundancy and initiated a lazy push set. This observation suggests that Plumtree can be utilized in the communication layer of the framework, even if the groups are small.

The experiment construction, as described previously in this chapter, has some limitations.

The dummy application does not produce many broadcasts. In an execution round with $n$ hosts, up to $n$ declare messages and a single bind message are sent through the Plumtree protocol. The number of broadcasts required for Plumtree to construct a stable spanning tree could be higher than this setup's evaluation allowed it to become. However, as Plumtree operates as a plain gossip protocol in the first message rounds, this should only affect the RMR results and not the reliability.

Twenty seconds per execution round is a short amount of time. Loomos are typically used for longer than a few seconds. So, the experiments are not fully representative of a typical Loomo fleet's potential uses. A potential cause for the HyParView issue could be the little time the protocol has to build a stable membership. In [27, p. 11], the first 50 of 250 cycles in the experiments were 'used to ensure stabilization of the protocols.' In this evaluation, only the first five of twenty seconds were reserved for HyParView. Five seconds only allow for a single round of shuffle requests. During the complete execution time, at most four passive peer exchanges can be finished in total. A different interval for shuffle requests might build a stable membership in less time.

The HyParView protocol cannot be examined in more detail because of the missing logs. So, the evaluation does not show whether HyParView requires large groups or more time to stabilize. Further experiments could provide insights into the minimal requirements of the communication protocols and investigate the HyParView issue's underlying cause.

[7] Maybe, the HyParView parameters as used in Riak [28] could be used as a starting point in future experiments.

Due to the setup, the results of this evaluation have limited comparability to the Plumtree results in [27], where a cycle-based simulator has been used. Running this project's evaluation application in Mininet was less controlled than a cycle-based simulation.

The Plumtree component's broadcasts were sent concurrently with the messages of the HyParView protocol, instead of performing the separate tasks individually. Additionally, multiple broadcast messages were sent concurrently in this evaluation. The Plumtree experiments performed one broadcast per round, and the step was finished when every message belonging to this broadcast was delivered. The advantage of a cycle-based simulation is more fine-grained control and evaluation options. However, compared to cycles, the Mininet emulation resembles a real network more closely, as the actual operating system code is executed instead of a simulated environment that only mimics the behavior of a real setup.[16, 52]

In addition to the cycle-based approach to simulate network failures in [27], some hosts were disconnected from the network. As explained in the environment setup, packet loss rates per link have been used in this evaluation. This setup introduces a different semantic to network issues. For example, packet loss rates can affect broadcasts and control messages of Plumtree and HyParView alike. Instead of several hosts going offline and thus not sending or receiving any messages, random messages by active hosts are lost. However, as the effect of the link packet loss rates was insignificant compared to the number of hosts, the consequence of this different semantic could not be examined well.

The framework design can be counted as a success. The strength is that components can be changed easily. This flexibility could be useful for future projects. To remedy the shortcomings of this thesis' evaluation, a different membership protocol could be implemented and used instead of HyParView. This only requires instantiating the other implementation and injecting it into the Plumtree constructor. The rest of the application and framework code can be left unchanged. This change could show the effect of the membership protocol in the framework's results.

## 5.4 Conclusion

A dummy application has been used to evaluate the framework in a Mininet environment. The dummy application was executed multiple times in different network setups. Log messages were written by the Plumtree component during the execution. Four metrics were calculated from the log messages' results. The broadcast reliability is one of the metrics.

The framework implementation achieved low reliability. Reliability below 25 % in the majority of the groups prevents the implementation from being considered usable in small groups. With outcomes as depicted in Figure 5.5, it is apparent that the framework is not ready for production usage.

This result does not say that no CRDT framework based on Selective Hearing can be used stably and reliably in small groups.

The Selective Hearing operations do not know any members in the group. Selective Hearing simply relies on the communication layer and is thus independent of group size.

It can be assumed that the Plumtree algorithms already performed well beginning with small groups. This assumption is based on observations with the LDH metric results. These results indicate successful usage of Plumtree in small groups.

HyParView was designed for large groups. This protocol may require differently tuned parameters for small groups. [16] notes that the correct configuration for scaled-down experiments might not be clear. Different parameter values could stabilize the results of this framework. Alternatively, another gossip membership protocol could be required. HyParView's design for large groups might mean that the protocol requires many members to operate. Lastly, an error in the HyParView component could cause unreliable operation as well. Unfortunately, the evaluation does not reveal which – if any – of these HyParView issues is the case.

The framework architecture can be used sucessfully in another implementation. It has strengths such as a decoupled design. This design allows a component to be exchanged with a different implementation; for example, if another membership protocol is tested with the framework instead of HyParView. The architecture does have minor weaknesses, such as the usage of strings instead of byte arrays in the parameter hand-off between the layers.

This thesis can only provide a partial answer to whether a CRDT framework based on Selective Hearing can be used stably and reliably in small groups. Looking at the results altogether, it can be said that the framework architecture, the slightly adapted Selective Hearing mechanisms, Plumtree, and the implemented CRDTs can be successful components in a future CRDT framework. However, the question remains whether HyParView is a suitable membership protocol in groups with only a few members.

The following chapter concludes the thesis and suggests potential future work for this framework.

# 6 Conclusion and Future Work

This thesis aimed to investigate the potential for stable and reliable usage of a CRDT framework based on Selective Hearing [35] in the context of small groups of nodes. This framework should support the basic workflow with CRDTs. It should make the distribution of its CRDTs transparent, as it implements a middleware layer [53].

An object-oriented framework architecture, based on Selective Hearing [35], was designed in this thesis. It consists of four major components: a CRDT library, a Selective Hearing package, the Plumtree gossip protocol [27], and the HyParView membership protocol [26]. These components are loosely coupled. They depend on abstractions instead of concrete implementations; therefore, each component could be easily swapped. The designed framework architecture can be used in future projects.

The framework has been implemented in the Java programming language. Due to the implementation's use of Java object serialization, the implemented framework does not support interoperability between different systems. The implementation was evaluated with a dummy application that uses the framework. The metrics from the Plumtree introduction [27], reliability, RMR, and LDH have been used in this thesis' evaluation. Additionally, a metric to examine the data produced by the evaluation environment has been investigated.

The implemented framework does not achieve sufficient reliability in the evaluation. It can be concluded that the implementation is not suitable for the usage in a small group of autonomous robots. Some observations in the evaluation indicate issues with the HyParView package in the context of this project. These issues are likely to be caused by a suboptimal protocol configuration in the context of the tested groups' small size. Other sources for the HyParView issues could be an error in its implementation or that HyParView is not designed for small groups.

Future work could investigate this thesis' issue in the gossip membership protocol. Experiments with different HyParView configurations may be performed to answer whether HyParView parameters have to change for different network settings. For example, it could turn out that lower TTL values for random walks are required with higher packet loss rates. A similar tendency for TTL parameters might be observable with smaller group sizes. These experiments could provide interesting insights into the HyParView protocol performance and requirements. Alternatively, HyParView might be compared to other gossip membership protocols, such as Atomix [10]. This approach might show whether some protocols are more suitable in specific

scenarios. However, as protocols are designed with different intentions in mind, it might not be possible to come to a useful conclusion. Both approaches could require a new and reliable HyParView implementation. A comprehensive protocol and contract testing strategy should be applied during the package development in addition to other software engineering practices such as test-driven development (TDD).

Another opportunity for future projects is an enhancement of the CRDT library. Future projects may investigate the changes required to use delta-based CRDTs in this framework. Delta-based CRDTs combine the strengths of state-based and operation-based CRDTs. Such a project would be interesting because autonomous robots would benefit from the lower bandwidth usage of delta-based CRDTs. Alternatively, future work might add more CRDTs and remove technical debt from the library. Instead of relying on the library implemented in this thesis, a future project could implement a framework based on an existing JSON CRDT such as Yjs [41] or Automerge [20]. However, implementing a full-featured CRDT library would probably not provide many insights, although required for projects that intend to use this framework in a production environment. Of course, enhancing the CRDT framework is not fruitful if the issue with the communication layer persists.

Future projects should be considered as the opportunities with CRDTs, and autonomous robots have the potential to outweigh the risks of these technologies. CRDTs can become the foundation of local-first software [21] – a user-friendly alternative to a cloud-first approach. However, CRDTs might also be used to serve ads better (as in the showcase example of [35]) or collect personal data about users. Autonomous robots can investigate catastrophe areas for dangers before humans enter. However, they might also be developed as autonomous war drones. Autonomous robots might also harm the environment because they consume energy, require rare earth elements and might become waste. The possibility to use technology for bad things does not mean it should not be developed at all. However, technologists cannot simply ignore the harmful effects of technology. Instead, we should strive to "do as much good as we can while doing as little damage as possible."[56]

# Bibliography


[1] Almeida, Paulo S. ; Shoker, Ali ; Baquero, Carlos: Delta state replicated data types. In: *Journal of Parallel and Distributed Computing* 111 (2018), Januar, S. 162–173

[2] Baquero, Carlos ; Preguiça, Nuno: Why Logical Clocks Are Easy: Sometimes All You Need is the Right Language. In: *Queue* 14 (2016), Februar, Nr. 1, S. 53–69. – URL https://doi.org/10.1145/2898442.2917756. – ISSN 1542-7730

[3] Blau, Taylor: *Verifying Strong Eventual Consistency in δ-CRDTs*, University of Washington, Bachelor's Thesis, Juni 2020

[4] Bourgon, Peter: *Consistency without Consensus: CRDTs in Production at SoundCloud*. Feb 2015. – URL https://www.infoq.com/presentations/crdt-soundcloud/. – Zugriffsdatum: 2021-03-18. – Recording and slides available from QCon.

[5] Bracha, Gilad: Generics in the Java Programming Language. (2004), 01. – URL https://www.oracle.com/technetwork/java/javase/generics-tutorial-159168.pdf. – Zugriffsdatum: 2021-07-07

[6] Bracha, Gilad ; Odersky, Martin ; Stoutamire, David ; Wadler, Philip: Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In: *SIGPLAN Not.* 33 (1998), Oktober, Nr. 10, S. 183–200. – URL https://doi.org/10.1145/286942.286957. – ISSN 0362-1340

[7] Brause, Rüdiger: *Betriebssysteme*. Springer Vieweg, 2017. – URL https://www.springer.com/de/book/9783662540992. – Zugriffsdatum: 2021-06-10. – ISBN 978-3-662-54100-5

[8] Brewer, Eric: CAP Twelve years later: How the "Rules" have Changed. In: *Computer* 45 (2012), 02, S. 23–29. – URL https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/. – Zugriffsdatum: 2021-05-11

[9] Comer, Douglas: Ubiquitous B-Tree. In: *ACM Comput. Surv.* 11 (1979), Juni, Nr. 2, S. 121–137. – URL https://doi.org/10.1145/356770.356776. – ISSN 0360-0300

[10] Contributors, Open Networking F.: *Atomix*. – URL https://atomix.io/. – Zugriffsdatum: 2021-05-11

[11] Demirbas, M. ; Leone, M. ; Avva, Bharadwaj ; Madeppa, Deepak ; Kulkarni, S.: Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases, URL https://cse.buffalo.edu/tech-reports/2014-04.pdf. – Zugriffsdatum: 2021-05-11, 2014

[12] Fowler, Martin: *Inversion of Control Containers and the Dependency Injection pattern*. Januar 2004. – URL https://martinfowler.com/articles/injection.html. – Zugriffsdatum: 2021-07-01

[13] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass : Addison-Wesley Professional, nov 1995. – ISBN 978-0201633610

[14] Gibbs, Samuel: *Google reinforces undersea cables after shark bites*. 2014. – URL https://www.theguardian.com/technology/2014/aug/14/google-undersea-fibre-optic-cables-shark-attacks. – Zugriffsdatum: 2015-05-24

[15] Gilbert, Seth ; Lynch, Nancy: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. In: *SIGACT News* 33 (2002), Juni, Nr. 2, S. 51–59. – URL https://doi.org/10.1145/564585.564601. – ISSN 0163-5700

[16] Handigol, Nikhil ; Heller, Brandon ; Jeyakumar, Vimalkumar ; Lantz, Bob ; McKeown, Nick: Reproducible network experiments using container-based emulation. In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies - CoNEXT '12*, ACM Press, 2012

[17] Inc., Segway R.: *Segway Robotics Developer*. 2019. – URL https://developer.segwayrobotics.com/. – Zugriffsdatum: 2021-04-23

[18] Jahns, Kevin ; Contributors: *Yjs*. – URL https://github.com/yjs/yjs. – Zugriffsdatum: 2021-05-11

[19] Kleppman, Martin ; Contributors: *Automerge*. 2021. – URL https://github.com/automerge/automerge. – Zugriffsdatum: 2021-05-11

[20] Kleppmann, Martin ; Beresford, Alastair R.: A Conflict-Free Replicated JSON Datatype. In: *IEEE Transactions on Parallel and Distributed Systems* 28 (2017), April, Nr. 10, S. 2733–2746

[21] Kleppmann, Martin ; Wiggins, Adam ; Hardenberg, Peter van ; McGranaghan, Mark: Local-First Software: You own your data, in spite of the cloud. In: *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ACM, Oktober 2019 (Onward! 2019), S. 154–178

[22] Lamport, Leslie: The Part-Time Parliament. In: *ACM Trans. Comput. Syst.* 16 (1998), Mai, Nr. 2, S. 133–169. – URL https://doi.org/10.1145/279227.279229. – ISSN 0734-2071

[23] Lamport, Leslie: Paxos Made Simple. In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), December, S. 51–58. – URL https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[24] Lantz, Bob ; Contributors, Mininet P.: *Mininet: Rapid Prototyping for Software Defined Networks.* – URL https://github.com/mininet/mininet. – Zugriffsdatum: 2021-05-26

[25] Lantz, Bob ; Heller, Brandon ; McKeown, Nick: A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks.* New York, NY, USA : Association for Computing Machinery, 2010 (Hotnets-IX). – URL https://doi.org/10.1145/1868447.1868466. – ISBN 9781450304092

[26] Leitao, J. ; Pereira, J. ; Rodrigues, L.: HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast. In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, June 2007, S. 419–429. – ISSN 2158-3927

[27] Leitão, João ; Pereira, José ; Rodrigues, Luís: *Epidemic Broadcast Trees.* May 2007. – URL https://asc.di.fct.unl.pt/~jleitao/pdf/srds07-leitao.pdf. – Zugriffsdatum: 2020-09-17

[28] Leitão, João ; West, Jordan: *Data Dissemination: From Academia to Industry.* 2014. – URL https://speakerdeck.com/jrwest/data-dissemination-from-academia-to-industry. – Zugriffsdatum: 2021-05-05

[29] Martin, Robert C.: OO Design Quality Metrics. (1994), Oktober. – URL https://linux.ime.usp.br/~joaomm/mac499/arquivos/referencias/oodmetrics.pdf

[30] Martin, Robert C.: The Dependency Inversion Principle. In: *C++ Report* (1996), Mai. – URL https://web.archive.org/web/20110714224327/http://www.objectmentor.com/resources/articles/dip.pdf. – Zugriffsdatum: 2021-07-01

[31] Martin, Robert C.: The Interface Segregation Principle. In: *C++ Report* (1996), Juni

[32] Martin, Robert C.: The Open-Closed Principle. In: *C++ Report* (1996), Januar

[33] Martin, Robert C.: *Design Principles and Design Patterns.* 2000. – URL https://web.archive.org/web/20150906155800/http://www.objectmentor.com/

resources/articles/Principles_and_Patterns.pdf. – Zugriffsdatum: 2021-05-19

[34] Martin, Robert C.: *Solid Relevance*. oct 2020. – URL http://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance. – Zugriffsdatum: 2021-05-19

[35] Meiklejohn, C. ; Van Roy, P.: Selective Hearing: An Approach to Distributed, Eventually Consistent Edge Computation. In: *2015 IEEE 34th Symposium on Reliable Distributed Systems Workshop (SRDSW)*, Sep. 2015, S. 62–67

[36] Meiklejohn, Christopher S. ; Van Roy, Peter: Loquat: A framework for large-scale actor communication on edge networks. In: *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2017, S. 563–568

[37] Meyer, Bertrand: *Object-oriented software construction*. New York : Prentice-Hall, 1988. – ISBN 0136290493

[38] Navalho, David ; Duarte, Sérgio ; Preguiça, Nuno: A Study of CRDTs That Do Computations. In: *1st Workshop on Principles and Practice of Consistency for Distributed Data*, ACM, April 2015 (PaPoC 2015)

[39] Navalho, David ; Duarte, Sérgio ; Preguiça, Nuno ; Shapiro, Marc: Incremental Stream Processing Using Computational Conflict-Free Replicated Data Types. In: *3rd International Workshop on Cloud Data and Platforms*, ACM, April 2013 (CloudDP 2013), S. 31–36. – ISBN 9781450320757

[40] Neuwirt, Max: *Picture of a Loomo*. 2021. – in private conversation

[41] Nicolaescu, Petru ; Jahns, Kevin ; Derntl, Michael ; Klamma, Ralf: Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types. In: *15th International Conference on Web Engineering*, Springer LNCS volume 9114, Juni 2015 (ICWE 2015), S. 675–678. – URL http://dbis.rwth-aachen.de/~derntl/papers/preprints/icwe2015-preprint.pdf

[42] Ongaro, Diego ; Ousterhout, John: In Search of an Understandable Consensus Algorithm. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA : USENIX Association, Juni 2014, S. 305–319. – URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro. – ISBN 978-1-931971-10-2

[43] Oracle (Veranst.): *Java™ Platform, Standard Edition 8 API Specification*. 2021. – URL https://docs.oracle.com/javase/8/docs/api/index.html. – Zugriffsdatum: 2021-06-14

[44] Preguiça, Nuno: Conflict-free Replicated Data Types: An Overview. (2018), Juni. – URL https://arxiv.org/abs/1806.10254

[45] Rüdiger Möller: *fast-serialization*. Juli 2021. – URL https://github.com/RuedigerMoeller/fast-serialization. – Zugriffsdatum: 2021-07-08. – original-date: 2013-12-05T20:36:00Z

[46] : *Segway Loomo Robot*. 2020. – URL https://de-de.segway.com/products/segway-loomo-robot. – Zugriffsdatum: 2021-05-19

[47] Shapiro, Marc ; Preguiça, Nuno ; Baquero, Carlos ; Zawirski, Marek: A comprehensive study of Convergent and Commutative Replicated Data Types / INRIA. URL http://hal.inria.fr/inria-00555588/, Januar 2011 (7506). – Research Report

[48] Shapiro, Marc ; Preguiça, Nuno ; Baquero, Carlos ; Zawirski, Marek: Conflict-free Replicated Data Types / INRIA. URL https://hal.inria.fr/inria-00609399, Juli 2011 (RR-7687). – Research Report. – 18 S

[49] Shapiro, Marc ; Preguiça, Nuno ; Baquero, Carlos ; Zawirski, Marek: Conflict-free Replicated Data Types. In: *13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, Springer LNCS volume 6976, Oktober 2011 (SSS 2011), S. 386–400

[50] Shvets, Alexander: *Dive Into Design Patterns*. URL https://sourcemaking.com/. – Zugriffsdatum: 2021-05-17, 2020

[51] Spear, Narmox: *Mininet Topology Visualizer*. – URL https://t1p.de/5sme. – Zugriffsdatum: 2021-06-04

[52] Sreerangaraju, Sreevatsa: *Emulation vs. Simulation | by Perforce*. – URL https://www.perfecto.io/blog/emulation-vs-simulation. – Zugriffsdatum: 2021-06-22

[53] Steen, Maarten van ; Tannenbaum, Andrew S.: *Distributed Systems*. 3. URL http://distributed-systems.net. – Zugriffsdatum: 2021-05-11, 2017

[54] Thomas, David ; Hunt, Andrew: *The Pragmatic Programmer*. 1. URL https://pragprog.com/titles/tpp20/. – Zugriffsdatum: 2021-06-23. – ISBN 9780135957059

[55] Torgersen, Mads ; Ernst, Erik ; Hansen, Christian P. ; Ahé, Peter von der ; Bracha, Gilad ; Gafter, Neal: Adding Wildcards to the Java Programming Language. 3, Nr. 11, S. 97–116. – URL http://www.jot.fm/issues/issue_2004_12/article5. – Zugriffsdatum: 2021-07-08

[56] Waldeck, Carsten ; Waldeck, Samuel: *SHIFT Wirkungsbericht*. 2019. – URL https://www.shiftphones.com/downloads/SHIFT-wirkungsbericht-2019-05-10.pdf. – Zugriffsdatum: 2021-07-15

[57] Wickham, Hadley ; Stryjewski, Lisa: 40 years of boxplots / had.co.nz. URL https://vita.had.co.nz/papers/boxplots.html. – Zugriffsdatum: 2021-06-16, 2012. – Forschungsbericht

[58] Wilson, Greg ; Aruliah, D. A. ; Brown, C. T. ; Chue Hong, Neil P. ; Davis, Matt ; Guy, Richard T. ; Haddock, Steven H. D. ; Huff, Kathryn D. ; Mitchell, Ian M. ; Plumbley, Mark D. ; al. et: Best Practices for Scientific Computing. In: *PLoS Biology* 12 (2014), Jan, Nr. 1, S. e1001745. – URL http://dx.doi.org/10.1371/journal.pbio.1001745. – ISSN 1545-7885

[59] Workgroup, SPDX: *MIT License*. 2018. – URL https://spdx.org/licenses/MIT.html. – Zugriffsdatum: 2021-05-12

# A Imagery

Figure A.1: A Loomo, turned towards the reader, with its display "face" visible.[40]

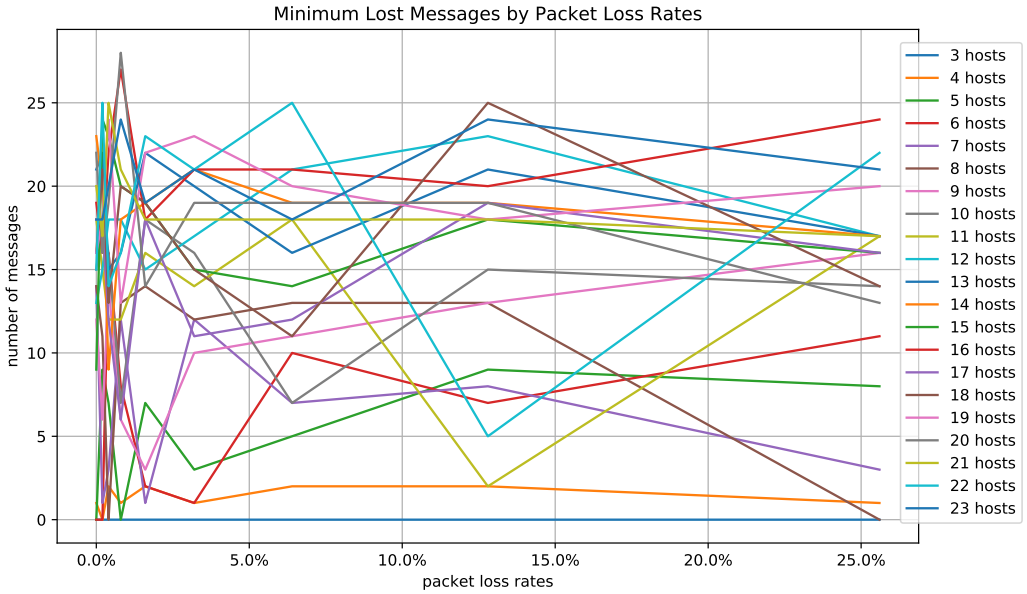


Figure A.2: Line chart showing the minimum lost messages by the used loss rates.

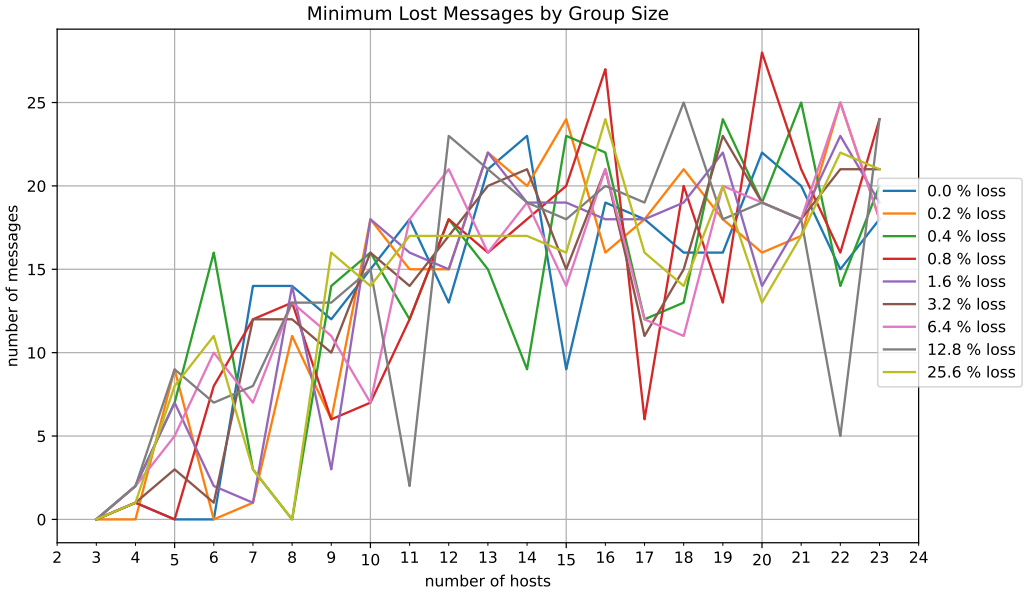


Figure A.3: Line chart showing the minimum lost messages by the used group sizes.

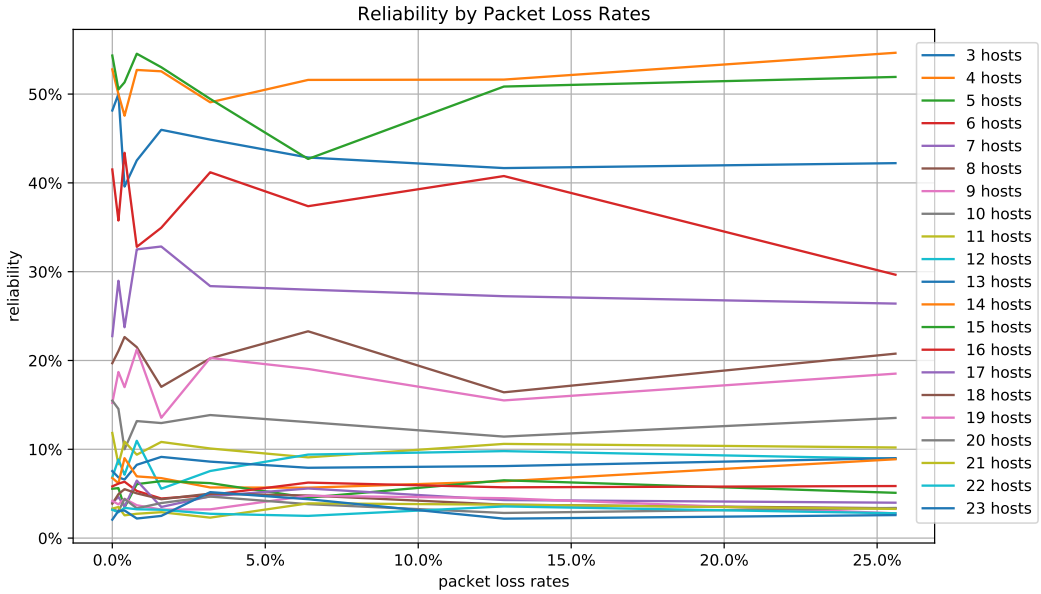


Figure A.4: Line chart showing the reliability by the used loss rates.

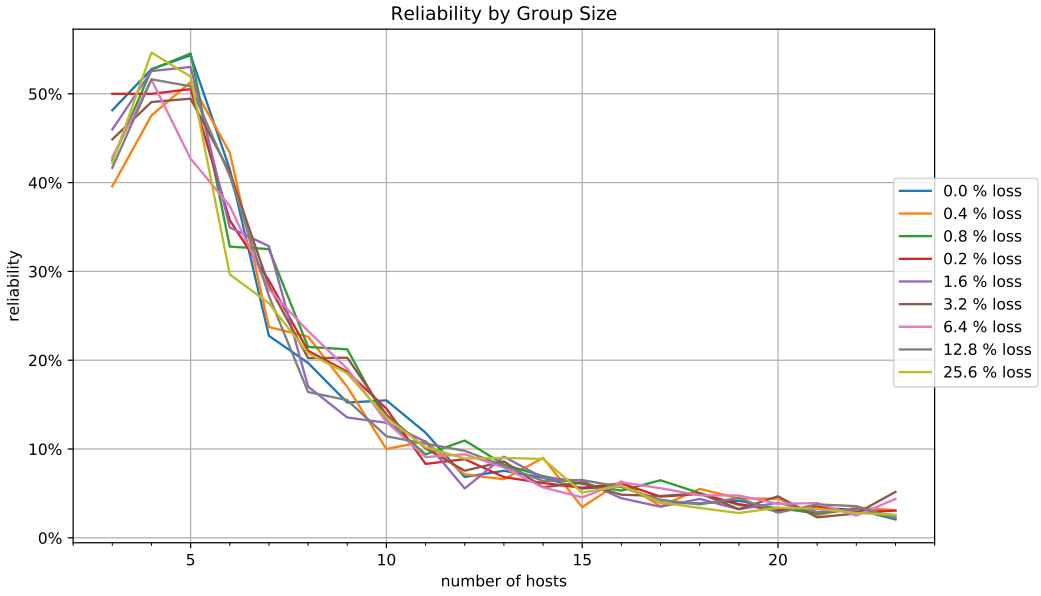


Figure A.5: Line chart showing the reliability by the used group sizes.

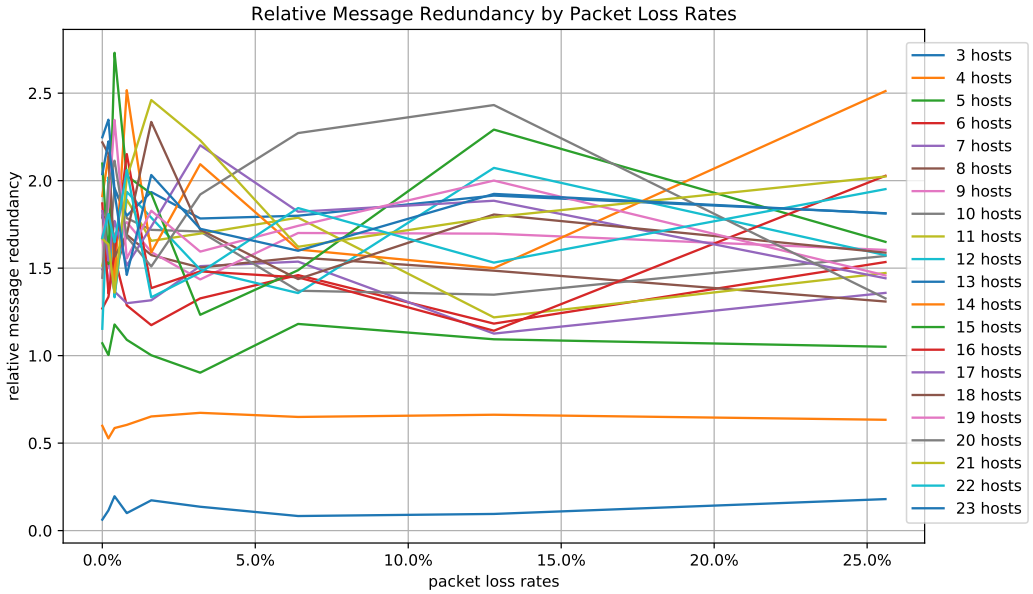


Figure A.6: Line chart showing the RMR by the used loss rates.

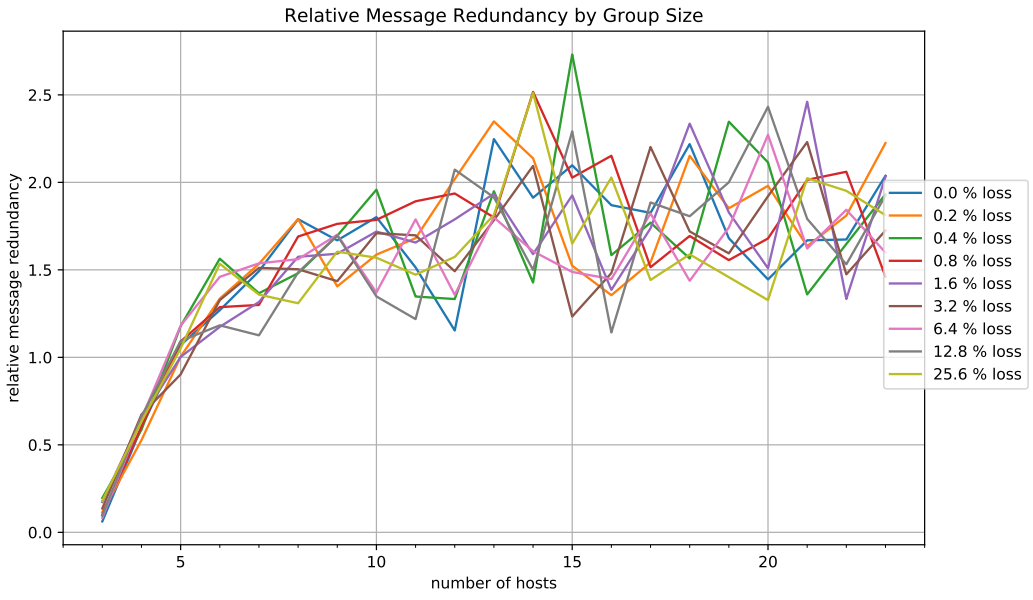


Figure A.7: Line chart showing the RMR by the used group sizes.

# Glossary

**CaDS group** A working group at the HAW Hamburg that focuses on distributed systems and general networking. xi, 1

**dependency injection [12]** In this pattern, a class allows its users to specify (inject) the concrete implementation of a dependency abstraction an object will use. 20, 28, 30, 37

**dependency inversion principle [30, 29]** This principle states: "High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces). Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions". 20, 26, 30, 37

**DRY principle** The 'do not repeat yourself' programming principle intends to make code easier to understand and maintain. It states that "every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [54, 58]. xi, 25

**eventual consistency** A weaker consistency guarantee, which allows temporary inconsistencies. 6, 7

**HAW Hamburg** The Hamburg University of Applied Sciences. xi, 1, 4

**HyParView** The Hybrid Partial View membership protocol [26]. xi, 6

**interface segregation principle [31]** The interface segregation principle produces small interfaces because it states that "clients should not be forced to depend upon interfaces that they do not use". 24, 37

**Lasp** A programming language implemented in Erlang providing deterministic coordination-free computation primitives based on CRDTs. 17

**Loomo** The Segway Loomo Robot is an advanced personal smart vehicle. xi, 1

**Mininet** "Mininet is a system for rapidly prototyping large networks on the constrained resources of a single laptop"[25, 24]. 44–47, 59

**open-closed principle [37, 32]** The 'open-closed principle' states "software entities should be open for extension, but closed for modification". 30

**Python** A programming language that is used in system administration, scientific projects, and many other areas. 44, 46, 50

**SOLID** A group of principles for object-oriented class design [33]. 4

**strong eventual consistency** A consistency model based on eventual consistency [48]. 2, 7, 8

# Symbols

**interest set** a set in Selective Hearing, containing all pending read requests to variables. 17, 18, 35, 36

**known values set** a set in Selective Hearing, containing variable identifiers with their latest observed values. 17, 18, 34–36

**known variables set** a set in Selective Hearing, containing the identifiers for all observed variables. 17, 18, 34–36

## Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

| Ort | Datum | Unterschrift im Original |
|---|---|---|