Degree Project in Information and Communication Technology

First cycle, 15 credits

Zero Self-View Latency: An Implementation of Conflict-free Replicated Data Types in Unity

A Benchmark of Operation-based CRDTs

TEOMAN KÖYLÜOGLU
JULIUS LARSSON

Zero Self-View Latency: An Implementation of Conflict-free Replicated Data Types in Unity

A Benchmark of Operation-based CRDTs

TEOMAN KÖYLÜOGLU

JULIUS LARSSON

Degree Programme in Information and Communication Technology
Date: June 26, 2025

Supervisor: Adam Hasselberg
Examiner: Paris Carbone
School of Electrical Engineering and Computer Science
Swedish title: Noll fördröjning i självvy: En Implementering av Konfliktfria Replikerade Datatyper i Unity
Swedish subtitle: En Prestandautvärdering av Operationsbaserade CRDTs

© 2025 Teoman Köylüoglu and Julius Larsson

Abstract

Interactive applications are deeply integrated in modern society, and form an integral part of everyday life. The need for low latency in these applications become more prevalent as technology standards and the scale of applications increase. This need for low latency is paramount in the field of Extended Reality (XR). Latency in XR becomes especially noticeable when a user's own actions are delayed. This concept is known as Self-View Latency (SVL). High SVL is known to induce Virtual Reality (VR) sickness, and reduce user immersion. One way to address this problem is to introduce Conflict-free Replicated Data Types (CRDTs) as part of a local-first computing paradigm for XR applications. These data types allow the user to apply local-first updates to achieve close-to-zero SVL, while guaranteeing conflict-free synchronization of states. In this thesis, we have identified data types that can be treated as CRDTs. We have investigated what support current networking frameworks have for CRDTs, and implemented a benchmark to measure the difference in latency. We have performed the benchmark in Unity using their own networking library Netcode for GameObjects (NGO). This benchmark compares a `Client-Server` topology, with a custom implementation of a `Distributed Authority`, and a custom `Relay-Server` that uses one CRDT, more specifically, the vector transformation of a sphere in Three-Dimensional (3D) space. The results of the benchmark showed that it is possible, for some operations that meet the requirements of CRDTs, to gain a close-to-zero SVL for the client that issued the operation. This extreme reduction in SVL additionally benefits all clients within a session, with potential implications for improving user immersion. Additionally, our benchmark shows that by implementing CRDTs, it is possible to reduce the total network traffic of the system. However, this effect diminishes as the number of clients increase. The number of clients used for the benchmark varied between 2 and 16. Further research is required for the implications of CRDTs scalability and performance in the context of XR applications. Lastly, research on how to effectively combine existing paradigms with CRDTs to optimize latency wherever possible, could also shed light on their practical use in a server-authoritative environment.

Keywords

Conflict-free Replicated Data Type(CRDT), Local-First Computing, Extended Reality(XR), Netcode, Strong Eventual Consistency(SEC)

Sammanfattning

Interaktiva applikationer är djupt integrerade i det moderna samhället, och spelar en central roll i det vardagliga livet. Behovet av låg latens i dessa interaktiva applikationer blir mer rådande då teknologiska standarder och applikationers skala ökar. Detta behov av låg latens blir av ytterst vikt i området för Utökad Verklighet (XR). Latens inom XR blir extra märkbart då en användares egna handlingar är fördröjda. Detta koncept är känt som Själv-Vy-Latens (SVL). Hög SVL är känt för att orsaka rörelsesjuka i Virtuell Verklighet (VR), och minskad användarimmersivitet. Ett sätt åtgärda detta problem är genom att introducera Konfliktfria Replikerade Datatyper (CRDT) som en del av en lokalprioriterad databehandlingsparadigm för XR-applikationer. Dessa datatyper tillåter användaren att applicera lokalprioriterade uppdateringar för att nå en nära-till-noll SVL, med en garanti för konfliktfri synkronisering av tillstånd. I denna tes, har vi identifierat datatyper som kan behandlas som CRDTs. Vi har undersökt det stöd som finns för CRDTs inom nuvarande nätverksramverk, och har implementerat en riktmärkning för att mäta latensskillnader. Vi utförde riktmärkningen i Unity med hjälp av deras egna nätverksramverk NGO. Denna riktmärkning jämför en klient-server topologi, med en självimplementerad distribuerad auktoritetstopologi samt en självimplementerad reläserver som använder en CRDT, mera specifikt, en vektortransformation av en sfär i 3D-rymd. Riktmärkningens resultat visar att det är möjligt, för vissa operationer som uppfyller kraven för CRDTs, att nå en nära-till-noll SVL för klienten som utförde operationen. Denna extrema reduktion av SVL har även nytta för alla klienter i en session, med potentiella implikationer för upphöjd användarimmersivitet. Dessutom, visar vår riktmärkning att det är möjligt, genom en implementering av CRDTs, att reducera ett systems totala nätverkstrafik. Å andra sidan, avtar denna effekt då antalet klienter ökar. Antalet klienter som användes var mellan 2 och 16. Vidare forskning krävs för att kunna urskilja implikationerna av CRDTs skalbarhet och prestanda inom området av XR-applikationer. Slutligen, forskning kring hur man effektivt kan kombinera existerande paradigmer med CRDTs för att optimera latens vartän möjligt, kan också kasta ljus på deras praktikalitet i en serverauktoritär miljö.

Nyckelord

Konfliktfri replikerad datatyp(CRDT), Lokalprioriterad Databehandling, Utökad verklighet(XR), Nätkod, Stark Eventual Konsistens(SEC)

Acknowledgments

We would like to thank our supervisor Adam Hasselberg for having guided us throughout this project. His regular involvement and continuous advice, particularly regarding experimental approach provided us with great tools and knowledge on how to address key aspects of our work.

We would also like to thank our examiner, Paris Carbone, for accepting us as his undergraduate thesis students, and inspiring us to pursue a thesis in this field. His knowledge, expertise, and deep engagement to the subject matter was invaluable to the development of this work.

Stockholm, June 2025
Teoman Köylüoglu Julius Larsson

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Problem . . . 3
1.2.1 Research Question . . . 3
1.2.2 Hypothesis . . . 4
1.3 Purpose . . . 4
1.4 Research Methodology . . . 4
1.5 Delimitations . . . 5
1.6 Structure of The Thesis . . . 5

2 Background **7**
2.1 Latency . . . 7
2.2 Topologies . . . 9
2.2.1 Client-Server Model . . . 9
2.2.2 Partial Ownership Topologies . . . 10
2.2.3 Peer-to-Peer Model . . . 11
2.2.4 Topologies Used in Unity Networking APIs . . . 11
2.3 Distributed Systems . . . 12
2.4 State Synchronization . . . 13
2.4.1 State Synchronization in Distributed Systems . . . 13
2.4.1.1 Strong Consistency . . . 14
2.4.1.2 Eventual Consistency . . . 16
2.4.1.3 Strong Eventual Consistency . . . 20
2.4.2 Conflict-Free Replicated Data Types . . . 21
2.4.2.1 State-Based CRDT (CvRDT) . . . 22
2.4.2.2 Operation-Based CRDTs (CmRDT) . . . 24
2.4.3 State Synchronization in Unity . . . 25
2.4.3.1 Remote Procedure Calls . . . 26
2.4.3.2 NetworkVariables . . . 27

2.4.3.3 Custom Messages 28
2.5 Related Work Area 29
2.6 Summary 30

3 Method **31**
3.1 Research Process 31
3.2 Research Paradigm 32
3.3 Data Collection 32
3.3.1 Sampling 32
3.3.2 Sample Size 33
3.4 Experimental design/Planned Measurements 33
3.4.1 Test Environment 33
3.4.2 Software To Be Used 35
3.5 Assessing reliability and validity of the data collected 35
3.5.1 Validity of method 36
3.5.2 Reliability of method 36
3.5.3 Data Validity 37
3.5.4 Reliability of data 37
3.6 Planned Data Analysis 37
3.6.1 Data Analysis Technique 38
3.6.2 Software Tools 38

4 Implementing CRDT Support in Unity **39**
4.1 Automatized Workflow 39
4.2 Unity Implementation 40
4.2.1 Client-Server 42
4.2.2 Simulated Distributed Authority 43
4.2.3 Relay-Server Incorporating CRDTs 44

5 Results and Analysis **47**
5.1 Major results 47
5.1.1 Analysis of Scatter Plots 47
5.1.2 Analysis When Combining The Scatter Plots 49
5.1.3 Analysis of Violin Plots 51
5.2 Hardware Setup 53
5.3 Reliability Analysis 54
5.3.1 Repeatability 54
5.3.2 Robustness 54
5.3.3 Error Sources 55
5.4 Validity Analysis 56

5.4.1 Internal Validity 56
5.4.2 External Validity 57
5.4.3 Construct Validity 58
5.4.3.1 A Mathematical Analysis for The RTT Distribution 59
5.5 A Closer Look at The Different RTTs 65
5.6 Potential Benefit of Using a Client-Host With CmRDTs 67
5.6.1 RTT Under a Client-Host Solution Leveraging CmRDTs 67
5.6.2 An Observed Benefit 69

6 Conclusions and Future work **73**
6.1 Conclusions 73
6.2 Limitations 76
6.3 Future work 78
6.4 Reflections on Sustainable Development 79

References **81**

A Scatter Plots 500 and 1000 Iterations **89**

B Scatter Plots 1500 and 2000 Iterations **97**

List of Figures

2.1 Difference Between Local-First Computing and Server-Authoritative Computing 8
2.2 Example of Rollback 26
2.3 A Server Issuing a ClientRPC 27
2.4 A Client Issuing a ServerRPC 27
2.5 A Server Synchronizing the Value of a NetworkVariable with Client 28

4.1 Program Workflow 40
4.2 Test Environment Overview 41
4.3 Client-Server RPC Workflow 43
4.4 Distributed Authority RPC Workflow 44
4.5 Relay-Server RPC Workflow 46

5.1 Benchmark Results for N=4 Clients and i=1000 operations . . 48
5.2 Benchmark Results for N=8 Clients and i=1000 operations . . 48
5.3 Average Delta t when increasing clients | i = 1000 50
5.4 Average Delta t when increasing clients | i = 2000 50
5.5 Violin Plot: All Topologies | n = 4, i = 1000 51
5.6 Violin Plot: All Topologies | n = 8, i = 1000 52
5.7 Violin Plot: All Topologies | Separated SVL | n = 4, i = 1000 . 52
5.8 Violin Plot: All Topologies | Separated SVL | n = 8, i = 1000 . 52
5.9 Client-Server RTT graph | Client-to-Client 59
5.10 Client-Server RTT Bucket Graph 60
5.11 Distributed Authority RTT graph | Host-to-Host 60
5.12 Distributed Authority RTT graph | Host-to-Client 61
5.13 Distributed Authority RTT graph | Client-to-Host 61
5.14 Distributed Authority RTT graph | Client-to-Client 61
5.15 Distributed Authority RTT Bucket Graph 62
5.16 Overflow of 0.5 RTT Values in Distributed Authority 63

5.17 Relay-Server RTT graph | Client-to-Self 63
5.18 Relay-Server RTT graph | Client-to-Other 63
5.19 Relay-Server RTT Bucket Graph 65
5.20 Difference between total RTT in topologies | i = 1 70
5.21 Difference in total RTT in topologies | n = 4 71
5.22 Difference in total RTT in topologies | n = 8 71

A.1 | Scatter Plot | 2 Clients 500 Operations 89
A.2 | Scatter Plot | 3 Clients 500 Operations 89
A.3 | Scatter Plot | 4 Clients 500 Operations 90
A.4 | Scatter Plot | 5 Clients 500 Operations 90
A.5 | Scatter Plot | 6 Clients 500 Operations 90
A.6 | Scatter Plot | 7 Clients 500 Operations 90
A.7 | Scatter Plot | 8 Clients 500 Operations 91
A.8 | Scatter Plot | 9 Clients 500 Operations 91
A.9 | Scatter Plot | 10 Clients 500 Operations 91
A.10 | Scatter Plot | 11 Clients 500 Operations 91
A.11 | Scatter Plot | 12 Clients 500 Operations 92
A.12 | Scatter Plot | 13 Clients 500 Operations 92
A.13 | Scatter Plot | 14 Clients 500 Operations 92
A.14 | Scatter Plot | 15 Clients 500 Operations 92
A.15 | Scatter Plot | 16 Clients 500 Operations 93
A.16 | Scatter Plot | 2 Clients 1000 Operations 93
A.17 | Scatter Plot | 3 Clients 1000 Operations 93
A.18 | Scatter Plot | 4 Clients 1000 Operations 93
A.19 | Scatter Plot | 5 Clients 1000 Operations 94
A.20 | Scatter Plot | 6 Clients 1000 Operations 94
A.21 | Scatter Plot | 7 Clients 1000 Operations 94
A.22 | Scatter Plot | 8 Clients 1000 Operations 94
A.23 | Scatter Plot | 9 Clients 1000 Operations 95
A.24 | Scatter Plot | 10 Clients 1000 Operations 95
A.25 | Scatter Plot | 11 Clients 1000 Operations 95
A.26 | Scatter Plot | 12 Clients 1000 Operations 95
A.27 | Scatter Plot | 13 Clients 1000 Operations 96
A.28 | Scatter Plot | 14 Clients 1000 Operations 96
A.29 | Scatter Plot | 15 Clients 1000 Operations 96
A.30 | Scatter Plot | 16 Clients 1000 Operations 96

B.1 | Scatter Plot | 2 Clients 1500 Operations 97
B.2 | Scatter Plot | 3 Clients 1500 Operations 97

B.3 | Scatter Plot | 4 Clients 1500 Operations 98
B.4 | Scatter Plot | 5 Clients 1500 Operations 98
B.5 | Scatter Plot | 6 Clients 1500 Operations 98
B.6 | Scatter Plot | 7 Clients 1500 Operations 98
B.7 | Scatter Plot | 8 Clients 1500 Operations 99
B.8 | Scatter Plot | 9 Clients 1500 Operations 99
B.9 | Scatter Plot | 10 Clients 1500 Operations 99
B.10 | Scatter Plot | 11 Clients 1500 Operations 99
B.11 | Scatter Plot | 12 Clients 1500 Operations 100
B.12 | Scatter Plot | 13 Clients 1500 Operations 100
B.13 | Scatter Plot | 14 Clients 1500 Operations 100
B.14 | Scatter Plot | 15 Clients 1500 Operations 100
B.15 | Scatter Plot | 16 Clients 1500 Operations 101
B.16 | Scatter Plot | 2 Clients 2000 Operations 101
B.17 | Scatter Plot | 3 Clients 2000 Operations 101
B.18 | Scatter Plot | 4 Clients 2000 Operations 101
B.19 | Scatter Plot | 5 Clients 2000 Operations 102
B.20 | Scatter Plot | 6 Clients 2000 Operations 102
B.21 | Scatter Plot | 7 Clients 2000 Operations 102
B.22 | Scatter Plot | 8 Clients 2000 Operations 102
B.23 | Scatter Plot | 9 Clients 2000 Operations 103
B.24 | Scatter Plot | 10 Clients 2000 Operations 103
B.25 | Scatter Plot | 11 Clients 2000 Operations 103
B.26 | Scatter Plot | 12 Clients 2000 Operations 103
B.27 | Scatter Plot | 13 Clients 2000 Operations 104
B.28 | Scatter Plot | 14 Clients 2000 Operations 104
B.29 | Scatter Plot | 15 Clients 2000 Operations 104
B.30 | Scatter Plot | 16 Clients 2000 Operations 104

List of Tables

2.1 Topologies supported by APIs
CS - Client-Server, CHS - Client-Hosted Server, RS - Relay-Server,
POT - Partial Ownership Topologies 12

5.1 Mathematical Formulae for RTT Distribution in the Different Topologies, RS - Relay-Server, DA - Distributed Authority . . 65
5.2 The total amount of RTT experienced by the different topologies 66
5.3 The different RTT buckets when using a Host with CmRDTs . 69

Listings

4.1 Empty MonoBehaviourScript 42

List of acronyms and abbreviations

2PC	Two-Phase Commit
3D	Three-Dimensional
ACP	Atomic Commit Protocols
API	Application Programming Interface
AR	Augmented Reality
CAP	Consistency Availability Partition Tolerance
CLI	Command Line Interface
CmRDT	Commutative Replicated Data Type
CPU	Central Processing Unit
CRDT	Conflict-free Replicated Data Type
CvRDT	Convergent Replicated Data Type
EC	Eventual Consistency
FIFO	First In, First Out
GB	Gigabyte
GDPR	General Data Protection Regulation
LAN	Local Area Network
MFU	Most Frequently Used
MR	Mixed Reality
ms	Millisecond
NGO	Netcode for GameObjects
P2P	Peer-to-Peer
RAM	Random-access Memory
RDT	Replicated Data Type
RPC	Remote Procedure Call
RTT	Round-Trip Time

S2PL	Strict Two-Phase Locking
SDG	Sustainable Development Goal
SEC	Strong Eventual Consistency
SVL	Self-View Latency
UDP	User Datagram Protocol
UI	User Interface
UN	United Nations
VR	Virtual Reality
XR	Extended Reality

Chapter 1

Introduction

The introduction chapter contains a brief background presented in Section 1.1. This is followed by the definition of the problem in Section 1.2. The aims are defined in Section 1.3. The research methodology used is presented in Section 1.4. Finally, delimitations are discussed in Section 1.5.

1.1 Background

Modern interactive applications require lower latency for a better user experience [1], [2]. These latency requirements become even more prevalent when looking at specific technologies, e.g. interactive applications like text editing tools and video games, or Extended Reality (XR) [1], [3], [4]. XR encompasses three different ways to experience extended reality: Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR) [5]. VR is a fully digital environment that replaces the real world around the user, AR adds a digital element, like an overlay to the existing world, and MR combines digital interaction elements with real world elements [5]. In other words, XR encompasses applications that support, augments, or interacts with the real world as part of its design.

Latency can be measured as the time it takes for communication between two endpoints in a network [4]. Examples of endpoints in this context are: a client, a server or a router. Networked applications are in general concerned with reducing the network delay between endpoints. In interactive applications, the delay perceived by the user becomes a more critical concern. A longer perceived delay will result in a worse user experience, as a client's own actions being delayed becomes more evident. Therefore, an important aspect

of interactive applications is the reduction of the latency perceived by the user i.e., Self-View Latency (SVL).

Over time, libraries have been introduced to help developers in the process of designing that pertains to latency. However, one problem with such libraries is that they often rely on existing paradigms whereby all traffic depend solely on the server routing and keeping track of game state [6]. The benefit of this topology is that the server acts a single source of truth; the server can seamlessly ensure consistent states across clients [6].

For current `Client-Server` topologies, when there are multiple clients in a session, all events must be routed and synchronized through the server [6]. This often comes at the cost of added latency, experienced as a delay by the client. One way to resolve this higher latency is to route traffic directly between clients instead of through a server. However, routing directly between clients removes the server as a single source of truth, and therefore requires a form of distributed state management.

One computing paradigm that enables distributed state management and changes the way data is synchronized and stored is local-first computing. In local-first computing, clients store and process data locally. The client does not have to rely on a server for these tasks. Storing and processing data between clients, without a server could therefore potentially reduce latency. However, this introduces new challenges with consistency of state. Under this local-first paradigm, copies of information are stored on different systems as instances of Replicated Data Types (RDTs). A certain RDT that addresses the previously mentioned consistency problem is Conflict-free Replicated Data Types (CRDTs), shown by Shapiro et al. [7]. A CRDT is a RDT that satisfies certain properties which guarantee that concurrent updates on an object may later be merged without conflict. CRDTs make use of a concept known as Strong Eventual Consistency (SEC), an extension of SEC, to ensure that multiple clients sharing the same set of updates are in an equivalent state [7]. This paradigm of local-first computing enforced by CRDTs therefore allows for seamless background synchronization between clients, with zero SVL.

In the design of interactive applications, game engines serve as a central component. Various game engines such as Unity, Frostbite, and Unreal Engine provide tools for developers to create and maintain large scale interactive applications. All of the aforementioned game engines enforce a server-

authoritative architecture where a server authority is used to maintain and synchronize states between clients. However, none of these engines enforce local-first computing principles in a serverless architecture. Recent research has explored the possibilities of implementing CRDTs in XR applications using a Peer-to-Peer (P2P) solution [2] for cooperation in a shared VR space. This paper will focus on examining how CRDTs can be implemented in current Unity networking frameworks and its potential gain. Unity, although not open source, was chosen as the main game engine for this thesis, as some of its networking libraries are open source and due to Unity providing adequate support for XR applications. Unreal Engine, like Unity, is partially open source, however, due to time constraints we chose to focus on one game engine.

1.2 Problem

We observe that current XR applications do not meet the fundamental user needs for low response time. The problem lies within the canonical assumption of server-authoritative architecture enforced by interactive XR application frameworks. This architecture forces all clients to synchronize their states through a main server. Due to the inherent latency involved within this architecture between the client and server, this becomes a limiting factor when considering scalability of large interactive virtual worlds. Therefore, we hypothesize that introducing converging data structures, such as CRDTs, in current XR application development frameworks will allow for distributed actions while offering zero SVL and eventual convergence across all nodes. To investigate this, we modify existing networking topologies in game engines to include support for CRDTs. Thereafter, we benchmark this implementation against established solutions to measure the benefit and cost of this approach.

1.2.1 Research Question

How can the implementation of CRDTs help reduce SVL in current XR computing frameworks used for making interactive applications?

What limitations are there in implementing CRDTs in current XR computing frameworks?

How could CRDTs be integrated seamlessly into existing XR computing frameworks?

1.2.2 Hypothesis

By inspecting interactive applications we observe that many operations can be moved outside the server and moved locally as to be CRDT operations. These CRDT operations will help reduce the SVL for all session users.

1.3 Purpose

The aims of this thesis is to bridge the gap between local-first and server-side computing, by examining the feasibility of incorporating CRDTs into existing XR computing frameworks. Additionally, to show that current networking topologies can be made more versatile and include multiple synchronization paradigms, or layers of transfer. These transfer layers would combine currently established topologies with local-first computing. Moreover, to show that CRDTs can be used to maintain consistency within such advanced frameworks. Furthermore, our aim is to reduce the latency for interactive applications by using this approach. Lastly, the aim is to investigate what possible functionality, limitations, and improvements, introducing a local first paradigm in current XR computing frameworks would yield.

1.4 Research Methodology

We will perform a literature review on current frameworks that are supported by Unity. We will check what functionalities these frameworks provide, more specifically the Application Programming Interfaces (APIs) that are used to synchronize states. We will evaluate this type of functionality to see which aspects of it can be replaced with CRDTs. We will then try to add the new suggested functionality through prototyping. We will define the scope of feasibility of implementing such functionality and its limitations. We will perform a comparative analysis of programs that use established topologies against our implementation of local-first computing, utilizing CRDTs.

As an overview, we will collect qualitative data, by performing a literature study, for the purpose of gathering information and contextualizing the problem area. The insights provided by the literature study will later help design an appropriate implementation framework. Thereafter, we will evaluate the efficiency of our suggested framework by collecting quantitative data for the purpose of comparing our suggested changes. This thesis will focus on both inductive and deductive reasoning.

1.5 Delimitations

Some of the server-side operations in a `Client-Server` topology cannot be moved to a local-first computing paradigm. This is due to the inherent constraint placed upon by CRDTs; CRDTs demand that these operations uphold certain properties that are not present in all operations [7]. This means that only a part of the server-side computation can be moved to a local-first paradigm. As a result, different interactive applications will benefit more or less from this approach. As such, the performative analysis and potential benefit of this study will not take into consideration this fact. Instead, focus will be placed on the feasibility of implementing CRDTs within existing networking frameworks, the improvements it makes in certain areas related to user experience, and the challenges that may arise.

CRDTs are not inherently secure [8]. This paper will not evaluate such an implementation from a cyber-security perspective. Furthermore, no ethical consideration will be taken in the field of cyber-law, more specifically data sharing, e.g., General Data Protection Regulation (GDPR).

1.6 Structure of The Thesis

Chapter 2 presents a literature study and the research conducted before designing the benchmark. Chapter 3 presents the method used to create the benchmark and the software used. Chapter 4 include the design of the program and how the benchmark was created with Unity. Chapter 5 presents the results, analysis of the validity and reliability of the results, and a discussion of the results. Chapter 6 concludes the thesis with limitations, future work and reflections.

Chapter 2

Background

This chapter contains the literature study and background information that was gathered for designing the benchmark. Section 2.1 explores how latency is perceived by users. Section 2.2 contains information about different topologies commonly used by Unity's networking frameworks. Section 2.3 defines what a distributed system is. Section 2.4 defines different consistency models, properties a CRDT requires, and how states are synchronized in Unity. Section 2.5 presents related work combining CRDTs and Netcode for GameObjects (NGO). Section 2.6 contains a summary of the background chapter.

2.1 Latency

Latency can be defined as the time it takes for information to propagate between different nodes in a network. In the context of interactive applications, latency can be thought of as the delay between user input and seeing the effect of said input. According to Cortés et al. [9], Self-View Latency (SVL) is defined as *"the difference between the time of a user's movement and the time when the user sees his/her move"* in the context of XR. Low latency is an important aspect of XR and interactive applications in general. Increased delay is associated with low user experience and immersion [9]. Therefore, one desirable outcome of an interactive application, is low latency, especially low SVL. An example of how a close-to-zero SVL can be achieved can be seen in Figure 2.1. This is accomplished by implementing local-first computing and therefore avoiding `Network Latency` that is inherent in a server-authoritative solution. As a client performs its local update it experiences no network delay, and thus sees its own update instantly.

In this paper, we define `Network Latency` as the part of the latency that is applied by the network transfer. We further define the `Synchroniaztion Latency` as the total latency for an update to take effect. Lastly, we define the `Overhead Latency` as the `Synchronization Latency - Network Latency`, the part of the latency that is not affected by the network transfer.

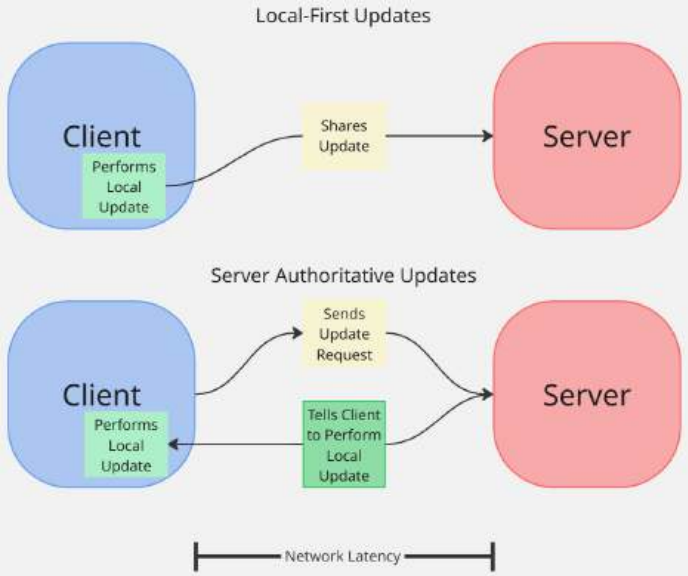

Figure 2.1: Difference Between Local-First Computing and Server-Authoritative Computing

An example of the difference between local-first computing and server-authoritative computing can be seen above at Figure 2.1. In the local-first scenario, the client would perform a local update, incurring a close-to-zero SVL and then broadcasting this update through the server to all clients. The SVL would in this case simply be a function of the `Overhead Latency`. In the server-authoritative scenario, the client would first send a request to the server to perform a given update. The server would then respond to the client (and all other clients in the session) to perform said update. The SVL would in this case be a function of the `Network Latency` and the `Overhead Latency`. We are aware, that in a literal sense, zero SVL is impossible due to `Overhead Latency` being unavoidable. However, in this thesis, we make no practical distinction between close-to-zero SVL and zero SVL and they are used interchangeably.

2.2 Topologies

A network topology refers to the way information is stored and sent within a system [10]. There are various types of topologies used in networking, each with their own benefits and drawbacks. We will begin this literature review by defining some terminologies, exploring some of the most commonly used topologies, and then proceed on to the different networking topologies used by Unity.

Round-Trip Time (RTT) is used to measure the time it takes for a package to transmit from a node, to some other node and back to the original node [11]. RTT is commonly used to measure the communication time between peers in a given topology, either between two distinct peers, or between a peer and server. This is beneficial because it measures the full time it takes for nodes to receive updates. This measurement takes into consideration the distance between nodes.

This RTT can also be thought of as a topological measurement instead of a scalar value denoting time, in which case this scenario just described would constitute 1 RTT. A packet propagating between any two nodes would therefore constitute 0.5 RTT, and a packet that is sent to a node and then transmitted back to the original sender, would topologically incur a 1 RTT.

2.2.1 Client-Server Model

In the `Client-Server` model, data is stored centralized with a dedicated server acting as the single source of truth [6], [12]. The central concept in the `Client-Server` model is that a client sends information to the server, the server then responds to all clients with the updated state [10]. In other words, the server is responsible for the storing, processing, and distribution of all data [12]. The main benefit of this model is that, since the server is in charge, it can easily provide more functionality, handle conflicts, and offer high performance [10]. The drawback of this model is the fact that all traffic must be routed through the server, i.e., if the server is farther away, the RTT increases [13]. The RTT varies among clients since the RTT depends on distance between the server and the client. The `Client-Server` model is used in various different software architectures [13].

The `Client-Host` model, similar to the `Client-Server` model, is a

model in which a server also acts as a single source of truth. However, the `Client-Host` model delegates one specific client to act as a session host [10]. The role of the session host is to act as a server for all other clients, while remaining a usable client. The benefit of this topology is a decrease in server maintenance costs. This is commonly used in different computer games, where a client hosts a session and acts as the server. The drawback of this topology is that the client that acts as server could act maliciously, e.g., cheat in video games. Additionally, the connection is limited to the network bandwidth of the host. Since residential connections may be slower than a server, this may result in increased latency [10], [14]. Meanwhile, the host is limited by the increase in computing power associated with being a host as well as a client [14].

2.2.2 Partial Ownership Topologies

In a partial ownership topology, ownership of objects is delegated to different clients. A client can act as a server authority for various objects in a session [15], [16]. Ownership can be distributed among clients to balance the workload. If a client wants to interact with an object, the client has to synchronize the action with the client that holds authority over that specific object or request ownership of that object [15]. If ownership is requested and accepted, the client becomes the new authority for that object. Different protocols can be used to change ownership of an object. For example, if a client interacts often with an object a Most Frequently Used (MFU) protocol could be used to determine which client should maintain authority over that object. However, if an object is frequently interacted with by various clients, constantly transferring ownership of said object could potentially prove inefficient.

In a virtual environment, where SVL would be more detrimental, this could result in a scenario where clients maintain ownership of the objects within their close proximity. This would allow them to update the objects without being affected by `Network Latency`. Therefore, the latency would be limited to the `Overhead Latency` of the system. This would then be credited to the fact that the client would simply synchronize the state of the object with itself, and thus eliminate the `Network Latency`. Since the client is now only left with the `Overhead Latency` this results in a close-to-zero SVL.

2.2.3 Peer-to-Peer Model

The P2P network topology is a form of distributed topology, where all clients in the network maintains the game state [17]. There is no single server by which all clients connect to, instead all nodes within the network communicate and share data with each other [18]. In other words, the P2P model is a distributed network topology where data is directly transmitted between clients without any intervening server. Since no server authority exists for this topology, nodes could potentially share malicious data to other nodes [18]. Therefore, this type of topology is more appropriate in smaller, trusted networks, for example Local Area Networks (LANs). Additionally, a P2P architecture could reduce server costs when comparing this solution to a server-authoritative topology [19]. However, when specifically applied to software architectures such as games, difficulties arise. More specifically, challenges emerge when additional constraints and properties are placed upon the transmitted data, leading to problems with synchronization [19]. These problems could be solved via the implementation of converging data types, such as CRDTs, to enable Eventual Consistency (EC) in a P2P topology.

2.2.4 Topologies Used in Unity Networking APIs

There are several networking framework APIs that have been developed for compatibility with Unity. Unity has their own networking API, NGO that has native support for three different topologies [10]. NGO supports `Client-Server`, and as part of the `Client-Server` topology a `Client-Host` solution is also supported. Lastly, `Distributed Authority`, a type of `Partial Ownership Topology` is also supported under NGO. Additionally, Unity Relay can be combined with NGO to create a `Relay-Server` [20]. A `Relay-Server` is a server that does not keep track of game state and simply forwards information between peers. Many third-party networking APIs have also been created with support for Unity. Some of the most popular frameworks include: `Photon Fusion`, `Mirror Networking`, `Steam Networking` and `Fish-Networking`. A comparison of the functionality of these frameworks will be done for the purpose of choosing an API framework with sufficient functionality for a benchmark.

`Photon Fusion` offers three different topologies: `Client-Server`, `Client-Hosted Server`, and `Shared` [16]; Shared is a type of `Partial Ownership Topology`. `Mirror Networking` offer two

different topologies, `Client-Server` and `Client-Hosted Server` [21]. `ISteamNetworkingSockets` support `Client-Server`, P2P transfer, and has support for a `Relay-Server` [22]. `Fish Networking` has support for `Client-Server` and `Client-Hosted Server` [23].

Table 2.1: Topologies supported by APIs
CS - Client-Server, CHS - Client-Hosted Server, RS - Relay-Server, POT - Partial Ownership Topologies

Framework	CS	CHS	P2P	RS	POT
Netcode for GameObjects	X	X	-	-	X
Photon Fusion	X	X	-	-	X
Mirror	X	X	-	-	-
ISteamNetworkingSockets	X	-	X	X	-
Fish-Networking	X	X	-	-	-

2.3 Distributed Systems

A distributed system consists of several independent nodes working in unison towards a common goal. A node could technically be a client, server or any kind of process. In this context, a node can usually be thought of as a server responding to requests from clients. These nodes would usually then be geographically placed such that clients would issue requests to servers in closer proximity. Additionally, the nodes of a distributed system share resources and pass messages through a network. To an outside observer or client, the distributed system appears as one single system [24]. In distributed systems, data is often replicated among all nodes, removing the single point of truth or failure. The advantage of this distributed topology, is that if one node shuts down, then clients can rely on other nodes in the system for their services. As such, availability and scalability is improved within distributed systems [25]. Another way that distributed systems improve on scalability issues inherent to centralized systems is to shard states. Sharding means to split some data into disjoint subsets, and in this context, these subsets are stored on different nodes. This reduces the fundamental computational load by splitting tasks between machines, providing increased scalability and performance. However, this has consequences for availability as losing one shard could result in the loss of critical data. As such, replicating data across several nodes could mitigate this problem and provide fault tolerance [26]. Nonetheless, sharding and replication has consequences for managing states,

and thus coordination mechanisms are necessary to ensure correctness and consistency across states [26], [27].

Referring back to Section 2.2.3, P2P is an example of a distributed system [25]. Each peer in P2P is an independent node and each node can perform tasks, request and serve data. However, in the context of distributed systems, the nodes are often thought of as several servers working in unison to achieve a shared goal. The servers are usually not compromised of clients, as in P2P, instead they are fixed servers distributing their workload between each other. Therefore, a more appropriate example of such is a distributed system is a `Distributed Client-Server` architecture. This topology is extremely common, especially in large scale enterprises. In such an architecture, servers are placed strategically to oversee traffic from specific regions, distributing traffic load and ensuring reliable performance for users.

2.4 State Synchronization

This section describes state synchronization in distributed systems, CRDTs, and state synchronization in Unity.

2.4.1 State Synchronization in Distributed Systems

According to Campêlo et al. *"a consistency model can be defined as a contract between a data storage system and the data processes that access it."* [28]. In other words, a consistency model provides some sort consistency guarantees for *"an update operation, as well as accessing an updated object."* [28] Consistency models are important as they allow us to reason about the state of our application given that we apply concurrent operations on shared objects where conflicts could emerge. This will further facilitate an understanding on the implications of implementing CRDTs on a system, more specifically the guarantees they may bring.

There are different types of consistency models discussed in distributed systems, for example: Strong Consistency, EC, and SEC. This section presents the differences between these levels of consistency, and highlights which consistency model is most suitable for this thesis.

2.4.1.1 Strong Consistency

Strong consistency, more formally known as linearizability, is a consistency model which guarantees the highest form of consistency, and is generally considered the most desirable one in terms of correctness. In a strongly consistent model, all reads on a shared object should always return the latest write [29]. In other words, clients should read the same object and never see different values. Additionally, non-overlapping operations on an object must be with respect to a real-time ordering. In other words, if an operation finishes before another, all clients must observe them in this order. However, overlapping (concurrent) operations can be ordered arbitrarily, as long as a consistent global ordering is applied. What's important here is that although these operations occur concurrently, the system still chooses a consistent ordering of these operations such that they appear sequential. This sequentiality must be consistent across all nodes, granting the illusion of a single ordered timeline [30].

Strong consistency can be understood in terms of two formal guarantees: linearizability and serializability [31]. Linearizability as defined in the previous paragraph, operates to "only allow executions whose results appear as if there is a single system image and 'global time' is obeyed" [32]. Serializability means that a concurrent execution, or a schedule, has an equivalent serial execution [33]. We can therefore see that linearizability and serializability apply at two different levels of abstraction, and both enforce total ordering. Linearizability pertains to total ordering of individual objects [34], whereas serializability means to schedule transactions such that they do not interleave and have an equivalent serial schedule [34]. To better understand why linearizability is closely related to serializability, it is important to understand *strict serializability*. Strict serializability refers to the concept in which the transaction upholds serializability, and the transaction history is sequentially ordered such that it upholds real-order execution [34]. As such, *"linearizability can be viewed as a special case of strict serializability where transactions are restricted to consist of a single operation applied to a single object"* [34].

Strong consistency uses these properties to give the impression that a distributed system acts as one single system. Strong consistency can be achieved under both centralized systems and distributed systems. In centralized systems, a common way to solve this is is via a single node

linearizing all operations; the node handles all traffic and verifies the order of operations and manages conflicts. To this end, locking schemes such as Strict Two-Phase Locking (S2PL) can ensure strong consistency [35]. This is beneficial because we know that the state will always be correct [7]. However, there are several issues that render strong consistency under centralized systems impractical, especially in the context of systems that require lower latency, scalability and availability. One problem with this type of structure is that the single node has to handle everything. The single source of truth can also act as a single source of failure, making the system vulnerable to different types of failure [7]; Power grid failures or cyber-attacks can disrupt the system since it requires the availability of one given node. Lastly, by scaling the system and thereby increasing the number of transactions that have to be routed through the node, the overhead is increased and users would experience latency [7], [31].

In distributed systems, sharding and replication necessitate the use of coordination mechanisms to achieve strong consistency. As mentioned in Section 2.3, sharding is well-suited for scalability and performance [26]. However, maintaining strong consistency across shards is challenging and requires coordination [36]. Atomic Commit Protocols (ACP)s such as Two-Phase Commit (2PC) ensure strong consistency by enforcing that all nodes must agree to commit, or none will [37]. The 2PC protocol consists of a transaction leader, the coordinator, and the cohorts, or participating nodes [27], [37]. Informally, in 2PC, the participants tell the coordinator they are ready to commit their transaction, and then await a response from the coordinator. If all participants voted to commit, then the coordinator will respond to all participants with an order to commit, otherwise it will order to abort. However, in this scenario the participants are waiting in a blocking state, and cannot release their locks until they receive an acknowledgment from the coordinator. This means that no resources held by this transaction can be used by other transactions while in this blocking state [37]. To mitigate this effect, timeouts may be used to recover in cases of indefinite blocking and communication failure [37]. However, this procedure further signifies the inherent limitations of 2PC. Despite their guarantee of strong consistency, the coordination strictness imposed by these protocols, as a result of sharding, incur added latency and reduce performance [36]. Additionally as mentioned in Section 2.3, sharding is not fault tolerant and may result in data loss. As such, 2PC is by itself not fault tolerant either and requires replication to solve the availability issue. However, this also has additional consequences

as the replicated data must now be coordinated as well [27]. In summary, to achieve strong consistency in a distributed replicated system, consensus based protocols are necessary [38]. Consensus is essentially the process by which the independent nodes of a distributed systems agree on a value, in other words, to commit on a single operation (value). Paxos and Raft are examples of common consensus based protocols [38] [26], and 2PC is a special case of a consensus protocol [37]. However, such protocols have significant overhead in terms of performance and have complex messaging systems to reach consensus [26] [37]. As a result, the level of coordination necessary in protocols for achieving strong consistency in distributed systems penalize latency heavily, and can thus be detrimental in interactive applications where latency is of high concern.

2.4.1.2 Eventual Consistency

EC is a consistency model in which the state of a distributed systems' nodes, or replicas, eventually converge to the same state, given that their updates have been delivered and updates are no longer shared [7]. More formally, it can be defined as upholding three conditions: *Eventual Delivery, Convergence, and Termination*.
According to **Definition 2.2** in [7], eventual delivery under eventual consistency can be defined as follows:

$$\forall i, j : f \in c_i \Rightarrow \Diamond f \in c_j \tag{2.1}$$

Informally, we can think of this definition as: For any replicas i and j, if a function f (method execution, applied at a replica) belongs to the causal history of c_i, meaning, if the function f is executed at replica i, then at some point in the future f (the update, or operation) will also belong to the causal history if c_j, i.e, it will be executed at replica j. In other words, this means that any update applied at a correct replica will eventually reach and be applied at all other replicas.

We will now explain the semantics used to define these formulas. The $\Diamond$ denotes that a given statement might hold now, or at some point in the future. In short, it can be explained as, for a given formula, say ϕ, where $\Diamond$ is applied onto ϕ such that $\Diamond\phi$, then ϕ holds now, or eventually at some point in the future, hence "eventual delivery". Perhaps as more familiarly known, the $\Diamond$ represents the symbolic expression for the F unary operator in temporal logic.

The i and j denotes replicas, and the $\forall$ in front means that it applies to all replicas.

The c_i denotes a causal history of a replica i. Simply put, a causal history of a replica i, is the set of all operations delivered (and applied) at the replica i, where the set also includes all causally preceding operations of any given operation within the set. For example, if an operation o_2 depends upon o_1, then o_1 must be included in the causal history for o_2 [7].

The term s_i and s_j denotes the states of replicas i and j. Following Shapiro et al. [39], state equivalence can be defined as the situation in which all read operations on i and j yields the same result.

Similarly, under the same definition (**2.2**) [7], convergence, the second criteria for EC is defined as:

$$\forall i, j : \Box(c_i = c_j) \Rightarrow \Diamond\Box(s_i \equiv s_j) \tag{2.2}$$

The $\Box$ is essentially the symbolic for the G unary operator in temporal logic, where a given statement must hold now, and for all future moments. Similarly then, as in (2.1) but with a stricter condition, $\Box\phi$ means that ϕ holds now, and for all future moments.

Intuitively, we can formulate this convergence definition as: If the causal histories c_i and c_j of any replicas i and j are equal at all moments from all future points in time, then eventually, their states will remain equivalent. In other words, if the replicas of a distributed system stop receiving updates, and correct replicas have delivered the same set of updates, then eventually their states will converge to an equivalent state [7]. The $\Diamond$ operator is important here as it reflects that convergence may be arbitrarily delayed with no guarantee of when it should occur, only that it will eventually happen. Consider two replicas that temporarily share the same set of updates but diverge at a later time, due to one receiving additional updates. If updates stop and their causal histories still differ, then eventual delivery (2.1) ensures that the missing updates are eventually delivered to all replicas. Only then will $\Box(c_i = c_j)$ hold, at which point convergence becomes guaranteed at some point in the future.

Lastly, termination simply means that *"all method executions terminate"* [7]. That is, the system must make progress and may not block indefinitely.

To further condition our understanding of EC, we can consider two types of properties: safety and liveness. We will come to see that EC is only a liveness property and not a safety property. A safety property can generally be understood as a certain invariant not violated during execution [40]. We will first use the definition of a property provided Alpern & Schneider in: [40]

Let s denote a program's state, s_0 denote the first state of the execution, and s_i denote the state at step i [40].

Let σ be a sequence of program states such that $\sigma = s_0, s_1, ..$ denotes an infinite sequence of said program states [40].
An execution does not in practice have to extend infinitely, and thus, a terminating execution is denoted by the terminating step infinitely repeating itself [40].

A program becomes a sequence of all possible executions σ that it can yield i,e. $Program = \sigma_0, \sigma_1, \ldots,$ likewise a property is a sequence of all possible executions σ, but only containing the executions that satisfy the property P [40]. In other words, a program that wishes to satisfy a given property must yield a subset of the property P: $Program \subseteq Property$. Simply put, all possible executions stemming from a program must be within the possible executions of the property P. If a given execution yielded from the program is not an execution with the set of P, then it has violated the property.

A safety property is usually, informally, defined as "nothing bad happens" during execution [40], [41]. A more exact definition of a safety property is provided by Alper & Schneider [40] as follows:

$$\forall \sigma : \sigma \in S^{\omega} : \sigma \notin P \Rightarrow \left(\exists i : i \geq 0 : \forall \beta \in S^{\omega} : \sigma_{[0..i]} \cdot \beta \notin P\right) \tag{2.3}$$

At this point, we only need to explain S^{ω}, and for simplicity let S^{ω} denote the set of all possible executions [40]. More precisely, this definition says, that for all partial executions σ belonging to the set of all possible executions, where the partial execution σ violates the safety property P, then there exists a finite point i after which all possible executions β concatenated with the partial execution σ violates the safety property P.
Informally, this definition says that if an execution violates a safety property P, then there exists a point i, from which all other possible continuations

beyond this point violates the safety property. This definition makes two important points; first, a safety property requires a finite prefix at which the property can be said to have been violated. Second, the definition also require that all possible executions continuing from the point i violate the property. Therefore, a safety violation can never be remedied after it has occurred [40]. In this definition of safety, it is easy to see that EC cannot be a safety property. In the definition for EC given at 2.4.1.2, only eventual convergence is guaranteed. It does not say anything regarding what must be true at every execution step. For example, if an invariant such as "no replicas will converge" is defined as the safety property, then at no point during execution can this property said to have been broken as it is possible they might converge sometime in the future.

A liveness property is essentially a positive promise about the future of an execution, or that a *"good thing happens during an execution"* [40]. In the case of EC a liveness property could be considered as the eventual delivery of all updates as expressed in (2.1). More formally, liveness is defined by Alper & Schneider [40] as follows:

$$\forall \alpha : \alpha \in S^* : (\exists \beta : \beta \in S^\omega : \alpha \cdot \beta \in P) \tag{2.4}$$

The expression S^* is defined as the *"set of finite sequences of program states"* [40].
This definition says that, for all executions α, where α belongs to the set of all possible finite executions (a partial execution), then there exists some execution β belonging to the set of all possible executions (a complete execution), such that α concatenated (followed) by β satisfies the property P. In other words, regardless of what has happened so far (α), then eventually there exists a future (β) where the complete execution ($\alpha \cdot \beta$) satisfies the liveness property P. It is then easy to see that EC is a liveness property, as it promises that something good must happen eventually, in this case eventual delivery. Notably, this condition must hold for all partial executions α, meaning that regardless of the state from the which state partial execution resumes, a promise of liveness can always be satisfied [40]. There are other, more restrictive, definitions proposed for liveness in the same paper [40]. However, for our purposes, it is enough to illustrate that EC is a liveness property.

In the context of designing distributed systems, it is desirable to uphold both

safety and liveness. Without safety, we can make no guarantee on the precise state of an execution, and without liveness we *"cannot stipulate that some 'good thing' always happens, only that it eventually happens"* [40]. As such, without safety, and only liveness, we cannot rule out any inconsistent behavior except that the good thing at some point actually happens. It is noteworthy to consider that EC first arose as result of consistency problems due to network partitions. A network partition is a situation in which nodes of a distributed systems are disconnected, and cannot communicate. In a distributed system, network partitions cannot be completely unavoided, and as such EC is a trade-off between consistency and availability in the presence of network partitions [41], [42]. The specific problem that occurs during a network partition is encapsulated by the Consistency Availability Partition Tolerance (CAP) theorem, which states that given a network partition, one must choose between either consistency or availability [41], [42]. As a result, to ensure availability during network partitions, lessening consistency in favor of availability captures the essence of what EC set out to achieve in the first place. As EC favors availability over consistency, it also enjoys the benefit of reduced latency, and clients need not wait for confirmation in order to serve local data [7]. Many modern applications have strict requirements on latency, as noted by the topic of thesis. This is one reason why, although safety cannot be guaranteed, EC still works well in many cases [7], [41]. The availability and reduction provided by EC in latency has proven to be an effective counterpart to strong consistency. If the time window between consistent states, and the number of old versions that are being read is scanty then an eventual consistent model can often behave in a strong consistent manner [41]. As EC is a rather weak model with no strict guarantees except eventual convergence, it is often considered a fundamental requirement, and specific data types can be combined with EC to bring forth stronger models that additionally guarantee safety [41].

2.4.1.3 Strong Eventual Consistency

SEC is essentially EC with the added guarantee of strong convergence. This property replaces the convergence defined in (2.2) with a better one. According to **Definition 2.3** [7] strong convergence is defined as follows:

$$\forall i, j \ : \ c_i = c_j \ \Rightarrow \ s_i \equiv s_j \tag{2.5}$$

Informally, this means that, for any two replicas i and j, if their causal histories c_i and c_j are equal, meaning if they have received the same set of updates, then their states s_i and s_j must be equivalent. The benefit of SEC is that is allows us to avoid certain inconsistencies from conflict resolution mechanisms that would otherwise occur under EC [7]. This also explains why it was necessary to define EC rigorously in 2.4.1.2, as it clarifies the precise distinction between EC and SEC. In this definition of SEC, the only difference between EC and SEC lies in how Strong Convergence is defined. The unary operators for convergence defined at (2.2), specify that convergence is only guaranteed at some point in the future once the same updates have been delivered at all replicas. It does not guarantee that they will immediately converge, and the implication here is that conflict resolution mechanisms may be involved in this process. In contrast, the ”strongness” of SEC lies in that convergence for replicas to which the same of updates have been delivered converge immediately. In other words, as soon as the causal histories of both replicas are equivalent, their states are already equivalent, with no further need for coordination.

Referring back to 2.4.1.1, normally, under a strong consistency model, replicas must coordinate (2PC, Paxos, Raft) before applying an operation, but as during a network partition coordination no longer is possible, some replicas must block to maintain consistency [43]. For this reason, SEC also provides a work-around for the CAP problem by ensuring availability even during network partitions without sacrificing consistency [7]. SEC renders this possible via the use of CRDTs that are inherently convergent. In other words, their operations incur no conflicts and converge deterministically without the need for coordination. The profound implication of this, as noted earlier, is that replicas are free to perform updates locally, in spite of network partitions, under the premise of conflict-free deterministic eventual convergence [7]. This renders SEC a strong contender for interactive applications where latency is of high concern.

2.4.2 Conflict-Free Replicated Data Types

One way to enforce SEC is by use of CRDTs. Indeed, the use of CRDTs—as formalized by Shapiro et al. in 2011—coined the term SEC [7]. A CRDT is a data type for which updates on replicas will converge to the same state, given that they share the same updates, without the need for conflict resolution. As such, CRDTs usefulness resides in their being conflict-free. Updates can be

applied immediately at the source and then simply broadcast to all replicas [7]. This renders CRDTs not only practical for scalability and in the presence of network partitions, but also significantly powerful in the pursuit towards zero SVL.

Lastly, in this section, we will now define three important mathematical properties that are of big relevance to CRDTs. We will come to see in the later sections 2.4.2.1 and 2.4.2.2, where we discuss two types of CRDTs in greater detail, how these play a role in upholding CRDT guarantees. The three mathematical properties are as follows: Commutativity, Idempotency, Associativity [44]. The first property is commutativity; Commutativity means that we can disregard the order of operands in a binary operation. Although we will discuss the *merge* function in more detail at Section 2.4.2.1, in the context of CRDTs, the join operation of the *merge* function of a state-based CRDT is an example of a commutative operation [7]. More intuitively, the $+$ operation is commutative since the order in which you add two numbers (operands) does not matter. The second property is Idempotency; idempotency means that the number of times an operation is applied does not affect the outcome [7]. As such, duplicate states or operations will have no effect. This property is important since reliable broadcast is not always guaranteed; if an operation, or state is shared twice, e.g., due to lack of acknowledgment, then it should not affect the result. We will talk about state-based CRDTs and operation-based CRDTs in greater detail at Sections 2.4.2.1 and 2.4.2.2, but for now it suffices to mention that, for state-based CRDTs, idempotency is upheld in its *merge* function, and in operation-based CRDTs it is upheld via a unique identifier attached to each operation (ensuring operations with the same identifier that have been shared twice is applied only once) [7]. The third property is associativity; associativity means that with a fixed order of operands, the outcome of the result is not affected by the position of the parentheses. For example, $(a \cdot b) \cdot c$ is equivalent to $a \cdot (b \cdot c)$.

2.4.2.1 State-Based CRDT (CvRDT)

Convergent Replicated Data Type (CvRDT) are a form of CRDTs where states are shared directly between nodes. The general form of a state-based CRDT, like any datatype, is a set of state replicas $s_i \in S$, where S is the set of all possible states [44]. Additionally, each object of a CvRDT has three operations; A *read* function, an *update* function , and the *merge* function [7]. The *read* function reads the state of an object and returns a value for the state.

The *update* function modifies the state of the object in a monotonic manner. Monotonic, in this context, means that values steadily increase with updates, and never decrease. The merge function is a join operation that makes use of an algebraic structure *join-semilattice*. The join-semilattice is a partially ordered set S, with a *join* operation whose join constitutes a *least upper bound* (LUB) for two merging states [7]. More formally, the join-semilattice can be defined as a structure $(S, \sqcup)$, where S is the set of all states, and $\sqcup$ is a join (merge) with the following properties:

$$\textit{Commutative: } a \sqcup b = b \sqcup a$$
$$\textit{Idempotent: } a \sqcup a = a$$
$$\textit{Associative: } (a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$$

The result of the join must constitute a *least upper bound* (LUB) of the merging states [7]. According to **Definition 2.4** [39] a *least upper bound* is defined in the following way: "$m = x \sqcup_v y$ is a Least Upper Bound of $\{x, y\}$ under $\leq_v$ iff

$$x \leq_v m \quad \text{and} \quad y \leq_v m$$

and there is no $m' \leq_v m$ such that

$$x \leq_v m' \quad \text{and} \quad y \leq_v m'."$$

Informally, this can be explained as taking two elements $\{x, y\}$ of a partially ordered set and joining them by an element m greater (or equal) than both x and y. Additionally, there must be no other element m' that is lesser (or equal) than m that satisfies this condition (hence the term *(least upper bound)*. Due to the inherent nature of partially ordered set, this means that the element m joins x and y in the sense that they inherit from m, or that m becomes the first common ancestor of x and y that joins them.

We now adopt the terminology *monotonic semilattice* described in [7]. In a monotonic semilattice, updates are always increasing in the semilattice with respect to the partial order [39]. For a state-based object to be recognized as a *monotonic semilattice*, it must have a *join semilattice* for its values, and for every update value in the semilattice, monotonicity has to hold. In other words, updates must always increase and converge towards the LUB [39].

According to **Theorem 2.1** (Marc Shapiro et al.) *"Assuming eventual delivery and termination, any state-based object that satisfies the monotonic*

semilattice property is SEC." [7]. Since the properties of the merge function upholds commutativity and idempotency, then regardless if messages arrive without order (commutativity), arrive multiple times (idempotency), the merge function handles it seamlessly due to its properties [39]. This puts less requirements on the communication channel, where the only requirement is that messages *eventually* reach. This makes a state-based CRDT practical when reliable broadcast is not an option.

In summary, for a state-based object to be recognized as a CvRDT (or simply a *state-based replicated datatype* (CRDT)) with SEC, it must fulfill the properties of the *monotonic semilattice*, eventual delivery and termination [7].

In a state-based CRDT solution, each replica has to share its entire state object. Since each replica sends its entire state (along with all its operations) to all other replicas, this entails a space complexity of $O(|operations| \cdot |replicas|)$ [44]. Although CvRDTs have weak requirements on the communication channel, this fact makes state-based CRDTs scale worse for larger applications and states.

2.4.2.2 Operation-Based CRDTs (CmRDT)

Operation-based CRDTs are in many ways similar to state-based CRDTs. In fact, there is a theorem that states that CvRDTs and Commutative Replicated Data Types (CmRDTs) are equivalent via emulation [7]. However, in practice and in theory, they have distinct characteristics that yield them practically different. In operation-based CRDTs, as opposed to sending the whole state, only operations are shared between replicas. Replicas receive operations and apply them locally for eventual convergence [7]. This reduces the network overhead as less data has to be sent across the network. Although the space complexity for CmRDTs still remains $O(|operations| \cdot |replicas|)$, as in the case of CvRDTs, the operation factor here is significantly smaller in comparison. As noted in the case of state-based CRDTs, in the case of unreliable broadcast, the merge method must be idempotent, commutative and associative. Unlike in state-based CRDTs, there is no merge method for the operation-based CRDT. This means that *reliable causal order delivery* is essential for CmRDTs to ensure ?? [7]. The operation-based object, like the state-based, has a *read* and a *update* function. The *ready* function is identical to the one described for the state-based [7]. However, the *update* function behaves slightly differently for the operation-based CRDT. The

update function is split into two operations; a *t* function known as a *prepare-update* function without any side effects. This is the operation applied at the source—a local-first operation—which in this thesis is the mechanism which would allow us to reach zero SVL for some operations. The second half of the update function, is the *effect-update* function which is executed at all replicas via a communication channel ensuring reliable causal order delivery [7]. In Unity, the local transform first issued by a client would represent the *prepare-update*, and the transform broadcast to all other clients is the *effect-update*.

The distinctive feature of the operation-based type is that a causal history is necessary for convergence. Therefore, if an underlying messaging protocol that supports *causal order delivery* is available for a given application, then an operation-based solution simplifies design significantly [45]. The following **Theorem 2.2** Marc Shapiro et al. summarizes the conditions for convergence in CmRDTs: *"Assuming causal delivery of updates and method termination, any op-based object that satisfies the commutativity property for all concurrent updates, and whose delivery precondition is satisfied by causal delivery, is SEC."* [7]. In the state-based CRDT, the condition of commutativity was placed upon the *merge* function. Here, commutativity is upheld by *reliable causal order delivery* and the *effect-update* function [45]. For trivially commutative operations, causal order delivery is not necessary, but is generally necessary since some operations require causal order for correctness [39]. An example of such a case where this is true, are the *add* and *remove* operations from an OR-set [39]. They are not inherently commutative, and as such they must arrive in causal order.

2.4.3 State Synchronization in Unity

Different networking libraries use different state synchronization models; different synchronization methods are implemented in combination with, e.g., rollback and client-side prediction to give the user a smooth experience. This further highlights how strong consistency is impractical in a game scenario where availability is of utmost importance and consistency may be reconciled at a later time. An illustration of rollback can be seen in Figure 2.2 where a client performs an update locally and proceeds to ask the server for permission. If the update is accepted, the client keeps the update, if the update is denied, the client reverts the update. Examples of these synchronization methods are Remote Procedure Calls (RPCs), `NetworkVariables`, and `Custom Messages` [46].

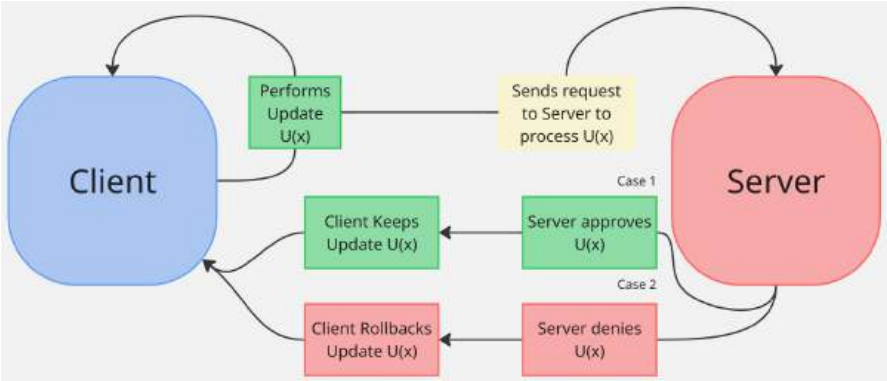

Figure 2.2: Example of Rollback

Unity, the proposed game engine for this thesis, uses User Datagram Protocol (UDP) and has native support for reliable delivery. As mentioned earlier, causal order delivery is generally required for CmRDTs. However, if a given CmRDT does indeed commute, then the causal order delivery restriction does not apply, and reliable delivery would suffice. More specifically, for causal order delivery two properties are required, First In, First Out (FIFO) and transitive order [47] [48]. A FIFO order can be defined as: Given an object o with a method X that supports remote execution, with a delivery time t, then if two events $X_1(o)$ and $X_2(o)$ occur, where $X_1(o)$ occurs before $X_2(o)$, then it must follow that $t(X_1(o)) < t(X_2(o))$. Lastly, Unity has support for implementing Unity Relay [49]. This renders operation-based CRDTs not only an interesting field of application, but a seemingly viable one.

2.4.3.1 Remote Procedure Calls

In NGO, RPCs are used for communication between two clients or between the client and the server authority. The purpose of an RPC is to be able to execute a method on a `NetworkObject` located on the server or another peer in the network [50], [51]. An example of a server sending an RPC to a client, telling a client to execute a method, can be seen in Figure 2.3. If several clients were connected to this session, they would also be a recipient of RPC, due to the ”Client” designation being a general one unless the RPC is specified to exclude certain clients. The destination of the RPC can be chosen from a wide variety of specific RPC method calls [52]. For example, a client can also issue a serverRPC seen in Figure 2.4.

RPCs in Unity’s NGO have reliability enabled by default with an option to switch to unreliable RPCs. In other words, *”this means they’re guaranteed*

Figure 2.3: A Server Issuing a ClientRPC

Figure 2.4: A Client Issuing a ServerRPC

to be received and executed on the remote side" [53]. However, this FIFO condition is upheld by RPCs in NGO when performed only on the same `NetworkObject` [53]. Despite this, the documentation makes no mention of transitivity. As such, we cannot conclusively state that NGO has support for reliable causal order delivery.

RPCs have native support for CmRDTs that trivially commute, e.g., vector transformations in Three-Dimensional (3D) space. But for RPCs to be suitable for all different CmRDTs, causal order delivery would have to be fulfilled. Since causal order delivery requires two properties, a FIFO order delivery and transitive order, an extension such as vector clocks could be implemented to achieve transitive order [45]. However, this might *"result in significant memory and message size overhead as the number of replicas grow"* [45].

Lastly, already transmitted RPCs will not be seen by clients joining after the RPC was sent, this could result in some de-synchronization between new clients and the current game states if RPCs were to be used for certain types of operations [53].

2.4.3.2 NetworkVariables

`NetworkVariables` are periodically synchronized between the server and clients in a `Client-Server` topology, or between clients and

object owner in a `Distributed Authority`. What differentiates `NetworkVariables` from RPCs is that they are in fact synchronized with clients after they join the session [54]. `NetworkVariables` support a preset of different variable types but can be expanded to include other variables as well. Examples of the preset types are e.g., `bool`, `byte`, and `int` [54]. An instance of a `NetworkVariable` can be seen in Figure 2.5 where the server sends the updated value of a dice to a client.

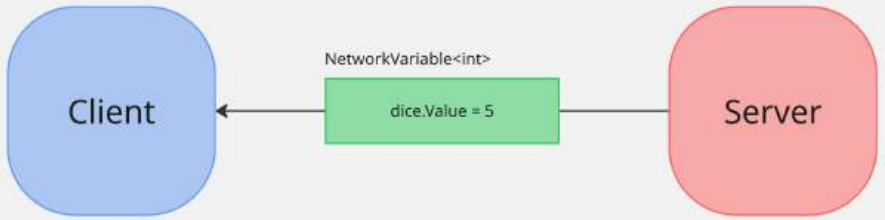

Figure 2.5: A Server Synchronizing the Value of a NetworkVariable with Client

When using a `NetworkVariable` for synchronizing across the network, a variable and a new value of that variable are chosen; a `NetworkVariable` can be updated as the value changes to ensure that new updates take effect, or at a set frequency to synchronize the value of all clients [54], [55]. Changes in `NetworkVariables` are not sent at the exact time they are issued. Instead, the changes are gathered between network ticks and are sent as the tick occur [55].

2.4.3.3 Custom Messages

Custom messages allow the user to define their own messaging system that deviates from the behavior of RPCs and `NetworkVariables`. It is not restricted to data types and allows custom functionality [56], which can be beneficial when dealing with i.e., reducing bandwidth by not sharing redundant information. Another useful area is one-way communication which can be very beneficial when implementing CRDTs in Unity's NGO. The flexibility of one-way communication in combination with broadcast messages and P2P messaging can enable custom messages a viable option for implementing CRDTs.

2.5 Related Work Area

This section will discuss the previous work done in this research area. Work has been done combining CRDTs with mobile applications evaluating at what extent operations can be moved to CRDTs for the purpose of collaborating with unreliable connectivity and offloading the server [57]. Studies have also been done in the field of online collaboration in spreadsheets, supporting offline collaboration with CRDTs [58]. Studies have also been done where CRDTs are introduced to vector graphics editing [59]. This is a field where some applications use a multi-player service for managing collaboration.

Furthermore, studies have been conducted on examining network delay of netcode in video games, more specifically rollback netcode [60]. Studies have also been done where CRDTs was created with functionality for gaming related tasks like leaderboards in mind [61]. Although some games like League of Legends have used CRDTs since 2014 [62], [63], not much research has been done regarding implementing CRDTs with current Netcode in popular API frameworks like Unity's NGO.

Studies have been performed examining the effect of SVL in combination with XR. One user study was conducted as a lab experiment where users completed tasks wearing a VR headset [9]. The experiment subjected the users to different levels of latency; the latency varied between 190 Millisecond (ms) and 597 ms. Users reported if it felt natural performing the tasks, if they experienced sickness, and the overall quality of the experience. The study showed that SVL is correlated with user experience from four metrics: Adaptation, Haptic Sensation, Global Quality of Experience, and Involvement. This study underscores the importance of low SVL in XR environments.

Lastly, not many studies have been done in the field of XR and CRDTs. A recent study by Carlos Baquero and Abel Dantas investigates the feasibility of using CRDTs in a decentralized P2P topology in a collaborative VR environment [2]. Interestingly, it also claims to possibly be the first instance of CRDTs being tested on VR. The study developed a prototype to benchmark latency and evaluate the synchronization of virtual objects within a shared VR environment. They test this approach first using operation-based CRDTs, and then state-based CRDTs. Although, the underlying method involves a strict P2P solution—besides the initial handshake via a WebSocket [45]—their work

proves highly relevant to our use case.

2.6 Summary

We will benchmark CmRDTs in unison with a `Client-Server` topology, where the server strictly relays information, to see if the performance of current topologies can be improved. The current topologies that will be tested against is `Client-Server` and a custom implemented `Distributed Authority`. The library that we will use is NGO, and the platform used is Unity. Due to time constraints, we decided to limit the scope of this thesis to Unity and to not include Unreal Engine. We decided to use NGO over other Netcode libraries due to them offering several topologies to benchmark against. Additionally, since NGO is an official library supported by Unity, its documentation standards and overall documentation are also closely aligned with Unity's. Unfortunately `Unity-Relay` and `Distributed Authority` were both pay-as-you-go. Therefore, we could not utilize the functionality of either `Distributed Authority` or `Relay-Server` provided by NGO. A custom implemented version of the benchmark was created for a `Distributed Authority` environment and a `Client-Server` topology was modified instead of using a `Relay-Server`.

The chosen operation for our CRDT is a vector transformation in 3D space of a sphere object. The operation will be to move the sphere towards one step in the `Z-axis`. A vector transformation in any direction is trivially commutative. To choose which type of CRDT makes the most sense for our implementation, understanding how idempotency would be upheld is crucial. As noted previously, in state-based CRDTs, idempotency is upheld by the *merge* function. However, in operation-based CRDTs, idempotency is upheld via ensuring that identical operations shared multiple times are only applied once. This condition is not explicitly said to be upheld by RPCs in NGO; therefore, a custom implementation to ensure idempotency would have to be implemented to ensure that this property is upheld. To solve this problem, we will track what messages have been already received. RPCs ensure reliable delivery and FIFO order as a delivery guarantee. Therefore, we chose a CmRDT that trivially commutes and will be using CmRDTs for the benchmark.

Chapter 3

Method

This chapter describes the research methodology used for designing the experiment and performing the benchmark. Section 3.1 describes a step-by-step overview of what was done to answer the research questions. Section 3.2 discusses the quantitative and qualitative parts of our thesis. Section 3.3 contains information about the sampling performed in the benchmark. Section 3.4 shows the process of creating the benchmark and the software that was used to achieve the results. Section 3.5 discusses reliability as well as validity of the data that we plan to collect as well as the research method that we plan to use. Section 3.6 contains the procedure that was used to analyze the data.

3.1 Research Process

The research process consisted of five steps. The first step was conducting a literature study. The purpose of the literature study was to gain familiarity with the current networking frameworks in Unity. More specifically, to discover what functionality and shortcomings current networking frameworks have. Additionally, the goal was to gain a deeper understanding of CRDTs and their different types. The second step consisted in choosing the most suitable networking framework in Unity; NGO was chosen in combination with CmRDTs as the most viable option. The third step was experiment design and implementing the benchmark. Three different topologies were chosen to measure the difference in latency and RTT between them. These topologies were implemented as separate scenes in Unity, with corresponding scripts to benchmark each scene. Scripts to automatize the experiment were created, in addition to process monitoring to streamline the experiments. The scripts

combine different benchmarking environments under one program, such that all primary data is generated through one parent process. The fourth step was running the experiment, this was done on a cloud server provided by KTH. Refactoring was done to allow the program to run on Linux for testing on a cloud server. The purpose of the cloud server is to run more clients and thus, increase the stress experienced by the system. The fifth step consisted of analyzing the data with mathematical formulae and reasoning about the synchronization in Unity.

3.2 Research Paradigm

We started with a qualitative literature study to learn about current networking frameworks and CRDTs. Thereafter, we prototyped a 3D sphere as a CmRDT in Unity which serves as the object of transformation in the benchmark. We created an automatized test environment for performing the experiment. We performed a quantitative experiment to test latency when introducing CmRDTs and local-first computing to current Unity topologies. The experiment benchmarks the difference in latency between a `Client-Server` topology, a custom `Relay-Server` using CmRDTs, and a custom implementation of `Distributed Authority`. This experiment was done under simulated conditions. We ensured simulated conditions to be able to control background variables. An artifact was designed to yield empirical data rather than using theoretical data. We chose this experimental research method over a systematic literature study because of interest in feasibility of implementation and latency comparison. Lastly, we chose to not perform a controlled user study since such an environment would introduce complexities that would not be needed for this type of study. Instead, we opted for automatization of the experiment to ensure control of background variables.

3.3 Data Collection

This section refers to how sampling was performed during the benchmark and what sample sizes that was used during the benchmark.

3.3.1 Sampling

The study is based on operation-based CRDTs, i.e., CmRDTs, in the context of 3D transformations in vector space. The benchmark was done to measure

latency between clients in different topologies. The number of clients and the amount of iterations varies between different iterations of the benchmark to include a more thorough data sampling and to ensure the internal validity of the benchmark. We are specifically interested in the effect that introducing CmRDTs have on SVL. Therefore, we measure the RTTs of packets when synchronizing states between different peers of the network. Lastly, we hope to observe any potential bottlenecks in any of the topologies.

3.3.2 Sample Size

The sample size, `n`, varies between 2 and 16 clients. That means that the number of clients used for a benchmark will be different for different iterations of the benchmark. The amount of iterations, `i`, varies between 500 and 2000, with increments of 500. An iteration in the context of this benchmark is an outgoing RPC issued by a client. 500 iterations means that every client will issue 500 RPCs to other clients. This will result in a total of $i \cdot n$ RPCs sent from users. A given RPC will be received by `n` users resulting in $i \cdot n \cdot n$ number of `End-Times` for each experiment. Each received RPC marks a difference in a `Start-Time` and `End-Time`, thus marking a Δt. Since $i \cdot n \cdot n$ are received by users, this means a total of $i \cdot n \cdot n$ Δts will be collected in each experiment.

3.4 Experimental design and Planned Measurements

This section describes the process that was used to create the Unity project and setting up the benchmark. The software to be used is also listed in this section.

3.4.1 Test Environment

Start with creating a unity 3D project. Install the following packages to your project:

1. com.unity.netcode.gameobjects
2. com.unity.transport
3. com.unity.multiplayer.tools

In the SampleScene, create an empty object and call it NetworkManager, and add the following components:

1. Unity Transport
2. Network Manager

Create an empty object and call it InteractableSphere. Add a Network Object component to the sphere. Press the Interactable-Sphere object, right click on it and create a 3D sphere object. Create a Scripts folder and create two scripts inside that folder, one Mono-Behavior Script called HeadlessLauncherClientServer, and one Network-Behavior Script called SphereControllerScriptClientServer. Add the launcher component to the NetworkManager, add the controller component to the Interactable Sphere.

The InteractableSphere and Network Manager objects are now part of a single SampleScene. However, for our benchmark, we need a scene for each topology and their corresponding HeadlessLauncher and SphereController scripts. As different scripts will act on the same scene, we need three builds for each topology, and as such we must duplicate this scene for each topology.

Inside the Assets folder, duplicate the SampleScene for each topology: Client-Server, Relay-Server, and Distributed Authority. Rename the scenes according to their corresponding topology names. Inside the Scripts folder, create subfolders for each topology and add both the HeadlessLauncher and SphereController scripts into the subfolders for each topology. Additionally, create another folder Editor. This folder will contain an additional BuildScript that builds each topology in headless mode for Linux or Windows.

Use RPCs to send transformations between clients. The RPCs should be sent with every network tick. Unity uses the Update() method to run ubiquitous code throughout the program. However, the Update() function runs on a significantly shorter interval than network ticks, therefore the Update() should be scheduled to send RPCs with every network tick instead. This will be performed in each of the HeadlessLauncher scripts by adding a delay inside of the Update() loop and only sending transformations when that delay is reached.

To determine the Δts for each synchronization the time must be measured. Measure the time before the RPC is sent, and when the same RPC is received by a client. Ergo, start the timer before each outgoing RPC, and end it with each incoming RPC. Since the clients and server run on one system, latency over the localhost can be introduced to be able to visualize the difference in RTT. For Linux this can easily be done in the Command Line Interface (CLI). For Windows, a tool like clumsy can be used. Use a one-way delay of 50 ms for the purpose of simulating a delay over the network. Perform the benchmarks with differing number of clients and amount of iterations (RPCs).

3.4.2 Software To Be Used

This implementation makes use of several tools and libraries. `PowerShell 5` is a minimum requirement. Furthermore, this implementation makes use of `Dotnet version 9`, and `Unity 6000.0.40f1`. Support for multiple operating systems have been implemented, therefore a user can use either Windows or Linux. However, note that to include Linux support, a cross-platform version of `PowerShell` is required.

If simulating on Windows, use Clumsy 0.3 for delay. The specifications with the built in port and local ip address will be covered with the command "`tcp.DstPort == 7777 or tcp.SrcPort == 7777 or udp.DstPort == 7777 or udp.SrcPort == 7777`". A 50 ms delay is preferred to noticeably see differences in RTT between the transactions. If simulating on Linux, a `PowerShell` script was created for introducing delay over `localhost`; this script is integrated with the `crossPlatformPerformSimulation.ps1` script.

3.5 Assessing reliability and validity of the data collected

This section contains the reliability and validity assessment of the data collection methods. The different design choices that were made to create the benchmark are explained and motivated.

3.5.1 Validity of method

To ensure validity of the chosen method of benchmarking, it is imperative that measured times are consistent. As such, time measurements are all performed on the same processor clock. This allows for synchronization and measurement of minuscule times, as small as microseconds, without discrepancy. Furthermore, changing only one variable at a time yields little room for interpretation. Cause-and-effect relationships are therefore given more clarity. For example, by only varying the number of clients or iterations, the effects of such an operation become clearly prominent.

The core of this benchmark is to compare a CmRDT solution to already established methods of synchronization. However, for such a comparison to be valid, we must use CmRDTs as the core data type for our transformation. A vector transformation in 3D space of a sphere upholds such a constraint. Furthermore, constraints also have to apply for the means of transportation. RPCs are used for all topologies as well as all types of synchronization steps in this test. Additionally, CmRDTs require that the mode of transformation upholds reliable causal order delivery. In other words, a client should apply updates in the order they were sent. RPCs do not guarantee causal order delivery and this is important to note. With this said, we have not implemented support for causal order delivery in this benchmark. This is because causal order delivery is not needed for a vector transformation in 3D space such as moving a sphere.

Running the test on a cloud server allows for close monitoring of the background processes and ensures that no external party interferes with the tests. Minimal background processes are used and are uniform throughout the test. This minimizes the margin of error.

3.5.2 Reliability of method

To ensure reliability of the testing environment, careful consideration of repeatability, consistency, and stability was taken when designing the benchmark. The test is fully automated to ensure that the margin of human error in the testing process is minimal. Furthermore, the benchmark uses the same builds for all iterations of the benchmark. Additionally, using large iteration values ensures that a holistic behavior of the system is acquired while disregarding outliers. Multiple instances of the test were run to ensure the repeatability of the test. In these tests, the number of clients

and number of operations vary. As few variables as possible are changed between the different instances of the test, ensuring that nothing interferes with the outcome. The amount of RPCs sent are limited to the network tick of the system, this reduces the impact of unforeseen traffic and yields more meaningful results.

3.5.3 Data Validity

To ensure the validity of the results, we opted to not only measure network ping but the time it takes for an update to take effect. We are interested in measuring how long it takes for operations to be applied across clients. One good measure of this is not only the internal time of the system, but also the RTTs between peers. Therefore, network delay was introduced to visualize the validity of the RTT. With the support of mathematical formulas we could derive how many times a message would be delivered over the network, depending on the sender and the receiver. To visualize this, we now refer back to the topological RTT defined in Section 2.2. As such, the RTT could be visualized as discrete values. If transfer over the network does not occur, the RTT is 0. If transfer occurs once, the RTT is 0.5. If transfer occurs twice, the RTT is 1. By increasing the network delay, the difference in RTTs will be more prominent and thus yield a partition of the Δts in the graphs. Via an induced delay on the network socket, the delay was ensured to be identical for all clients. Lastly, comparing the expected RTT with Unity documentation was done to further corroborate our findings.

3.5.4 Reliability of data

Reliability of the data could be maintained throughout the experiment with two specific requirements. The first requirement is that the test incurs the same induced latency for all different iterations. The second requirement is that similar results are achieved when running on different systems. Scatter plots were evaluated to ensure similar patterns, in different iterations of the test. Furthermore, when inducing a latency of $50ms$ we expect the value of the Δt to be over at least $100ms$ for operations with an inherent RTT of 1.

3.6 Planned Data Analysis

This section describes the procedure and technique for analyzing the data, as well as what tools were used to analyze the data.

3.6.1 Data Analysis Technique

The first step of the data analysis consisted of generating scatter plots to be able to differentiate trends in the data that was collected. The scatter plots consisted of plotting the time sent of the RPC on the `X-Axis` and the Δt on the `Y-Axis`. The Δt therefore represents the latency, and this would show how the latency would change over time. To make these trends more visibly distinguishable, the network delay was configured to introduce a latency sufficient to produce a clear trend of the RTT. For the purpose of gathering more insight into the data, violin plots were created to visualize distribution of the RTT. This yielded more insight into how the RTT was distributed in the overall manner of the test. To categorize the different measurements of the RTT, bar plots were created with the purpose of collecting discrete numbers for the different RTT values. The bar plots grouped the RTTs into buckets of 0, 0.5, and 1 RTT.

3.6.2 Software Tools

To analyze the data `Jupyter Notebook` was used to create graphs and compile the data into a larger dataset. The graphs created with `Jupyter Notebook` consisted of scatter plots, violin plots and bar plots.

Chapter 4

Implementing CRDT Support in Unity

This section covers an introduction to the overarching design of the benchmark. The benchmark was designed with automation in mind, such that all necessary data should be generated with the press of a button. To utilize this, we combined `PowerShell` scripting with C# scripts written for Unity. `PowerShell` scripting is used to manage the different processes generated by the C# scripts. Therefore, we can think of the `PowerShell` script as a parent process observing the child processes until they terminate.

4.1 Automatized Workflow

The program consists of many different scripts bundled into one script: `CrossPlatformSimulation.ps1`. When we refer to the program, we are referring to all the scripts contained and executed within `CrossPlatformSimulation.ps1`. The purpose of this script is to completely automatize the testing process. The script first defines a parameter list, that contains the amount of clients and iterations to be performed. Thereafter, it iterates through all topologies and for each topology iterates over the parameter list. A single benchmark instance for a given topology and parameter list is then conducted within each inner loop. The `$Topology$Script.ps1` monitors all created processes and waits for them to finish. For any given benchmark a topology-authoritative process is created. A given amount of clients is created for each test. A complete illustration of the workflow is included in Figure 4.1.

Figure 4.1: Program Workflow

The varying number of clients is the same for each topology. In `Distributed Authority` the host also acts as a client. For the remaining topologies, a server is not considered a client. Therefore, the number of processes created for `Client-Server` and `Relay-Server` are increased by one.

When all transactions are finished, a client writes primary data to two `.csv` files. The primary data file consists of an informational header for the specific benchmark in question. The primary data consists of the following information: `client ID, transaction ID, corresponding start-` and `end times`. These two files are abstracted within the `ClientN.csv` file for the purposes of this illustration. When all clients are done writing to their files, the `Topology Authority` shuts down all clients. Lastly, `ListMerger.cs` is executed by `Crossplatformsimulation.ps1`, generating the combined primary data into the `MergedFile$Topology$.csv` file.

4.2 Unity Implementation

The different topologies are, `Client-Server`, a custom `Distributed Authority`-like topology, and custom `Relay-Server` incorporating CmRDTs. In Unity, a separate scene is created for each topology, and as such, a build is created for each scene. The benchmarks of the different topologies are performed on each build. In other words, build once, run many times. What characterizes each scene is how the sphere is

transformed, since that changes depending on the current topology. All other aspects besides synchronization, such as connectivity, timestamps, and other redundancies, were minimized to simulate a consistent environment throughout all topologies. The environment is kept simple. The benchmark runs in `Headless Mode`, not simulating any graphics. An overview of the test environment can be seen in Figure 4.2. The benchmark consists of synchronizing the simple sphere that can be seen in Figure 4.2. The User Interface (UI) present in the figure is a developer overlay.

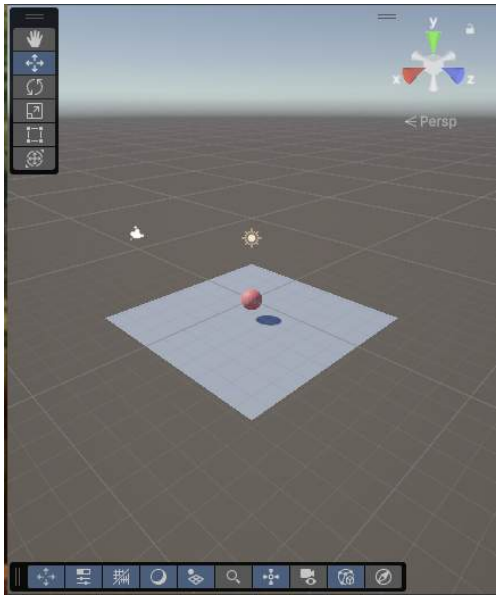

Figure 4.2: Test Environment Overview

Each individual test is designed in a way such that n clients run one operation scheduled at every network tick, resulting in a total of i operations. This means that, a total of $i \cdot n$ RPCs are sent from clients for a given test. To achieve this we use a `Mono-Behaviour Script`. An example of an empty script can be seen in Listing 4.1. The start function is only performed once at the start of execution. The update function is performed once after the start for every *frame*. This *frame* is not deterministic, and is therefore susceptible to variation. We use the `Update()` function to schedule our RPCs. Moreover, a network tick is a discrete interval in which Unity handle network-related events. In other words, this is the time frame in which RPCs are handled by the system. It should be noted that the `Update ()` function is called several times between any given network tick. Therefore, the `Update ()` function becomes problematic since it essentially bombards the network

with RPCs in a much shorter time frame than a single network tick. Since the interest of this study is latency and not throughput, it would be beneficial for the simulation to not be overrun with a surrealistic amount of RPCs. The `Update ()` function can be throttled by the help of a timer that tells the client to only send a RPC when the timer is depleted. In our case the timer was limited to the duration of a network tick. This allows us to send RPCs at a determined fixed rate of 30 RPCs per second, with a balanced traffic load.

Listing 4.1: Empty MonoBehaviourScript

```
{
  public class MonoBehaviorScriptExample : MonoBehaviour
  {
    void Start()
    {

    }

    void Update()
    {

    }
  }
}
```

4.2.1 Client-Server

In a `Client-Server` topology, the server handles all traffic and acts as a single source of truth. This fact remains true for the topology used in this benchmark as well. The server waits for all clients to connect; when all clients are connected and their startup is completed, they report back to the server. The server then tells all clients to start sending RPCs. The clients send one RPC per network tick until the amount of RPCs chosen for the test have been reached. When the clients have received all RPCs from all other clients, they report back to the server. The server then proceeds to tell the clients to write to their files. The clients report back to the server when the files have been written. When all files have been written, the server tells the clients, and itself, to shut down their processes. The RPC workflow, which can be seen in Figure 4.3 show that all RPCs are routed through the server. A client starts its `Start-Time` and sends a `ServerRPC` to the server. The server transforms the sphere on their end and sends a `ClientRPC` to all clients telling them

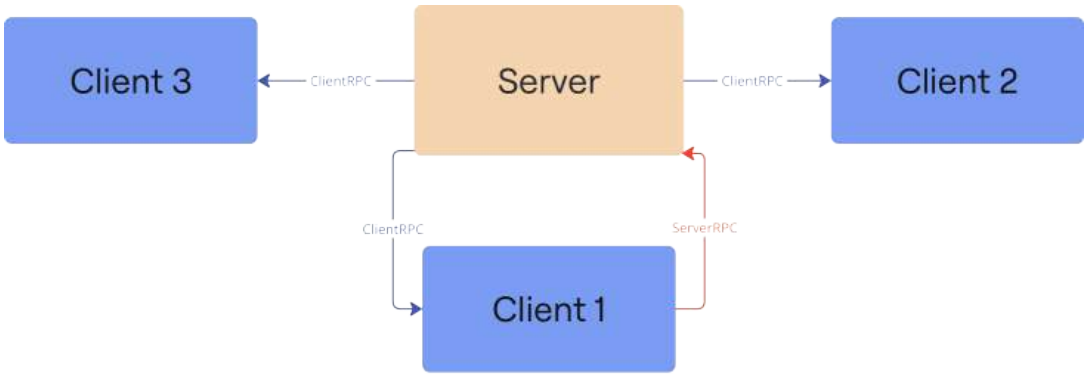

Figure 4.3: Client-Server RPC Workflow

to transform on their end, documenting the `End-Time`. The `Start-Time`, `End-Time`, `operation-id` and `client-sender-id` are all saved to a list. In this topology the server maintains the game state by performing its own transfer every time they receive an RPC from a client.

4.2.2 Simulated Distributed Authority

In a `Distributed Authority` the ownership of objects is divided between the clients, and can be dynamically transferred during runtime. This reduces the load on a single server or host acting as a single source of truth. The advantages of this, is that the stress on a single host can be divided between all peers. For example, when a client wishes to act on an object, it can either request permission to change the object, or to request ownership of said object. In the latter, if accepted, the requesting client will become the authority for that specific object.

In our benchmark, we are performing transformations on a single object and moving it for each transformation. This object can only belong to a single peer at a time. To assume that the object would change ownership at every operation, would not be a realistic scenario and may result in an unrealistically high latency. For this purpose, it makes more sense for the ownership of the object to reside with one client for this type of benchmark. All other clients would simply request permission to change the state of the object. However, this introduces a skewed representation of what `Distributed Authority` seeks to address. As a result, this topology resembles a `Client-Host`, or a `Distributed Authority` where ownership is never changed. We never change ownership of the object for this test, due to it

being performed on one single object multiple times during one test. In fact, using a true `Distributed Authority` solution would require paying for Unity Cloud services. As a result of these constraints, this solution can be seen as a simplified version of one scenario that happens in `Distributed Authority`. Since `Distributed Authority` is not inherently used for the benchmark but rather simulated, we aim to achieve a broader sense of this type of `Partial Ownership Topology` and not the specific topology provided by Unity. These results would examine the RTT concerning a broader topology network similar to that of `Distributed Authority`. On the other hand, latency under `Distributed Authority` provided by Unity will be more difficult to prove since it is not directly being used. Lastly, this data could be used to determine whether a client-based or host-based topology experiences a bottleneck first by looking at the increase in latency as we increase the number of clients.

Figure 4.4: Distributed Authority RPC Workflow

The way RPCs are sent in the simulated `Distributed Authority` can be seen in Figure 4.4. Client 1 notes the `Start-Time` and sends a ServerRPC to the host, who acts as both a server and a client. In turn, the host sends a ClientRPC to everyone, including itself. The clients receive the RPC and perform the transform. Lastly, the clients note the `End-Time` and saves all data gathered for this operation to a list. The host, would in our simulation of `Distributed Authority` incur two extra tasks as opposed to other clients. The host would be the owner of the sphere that is being transformed, and the host would be the session owner.

4.2.3 Relay-Server Incorporating CRDTs

The idea behind this topology is to enable a CRDT solution where clients perform their transforms locally and transmit their operations without acknowledgment from any centralized server. This raises an important question; how do we ensure communication between peers without a server? The answer to this question is to use a `Relay-Server` that simply forwards RPCs between clients during the benchmark. Another point to consider, is the

use of strict P2P between peers. However, in this scenario, several things have to be taken into account first.

One key aspect of this benchmark is to showcase that a game, normally using a server-authority, potentially could incorporate local-first computing as part of its design wherever possible. This means that a benchmark using CRDTs should in some sense work under similar conditions as the `Client-Server` solution. This renders true P2P a suboptimal solution as it significantly deviates from the aims of this thesis. More realistically, a lightweight `Relay-Server` would be more appropriate. In the `Relay-Server` topology a lightweight server manages two tasks. The first task is establishing connection for clients. The second task is relaying messages over the established connections. The server does not keep track of the game state in this topology but it is managed in another way; the game state is automatically managed through CRDTs and reliable delivery of updates. It should be noted that, in this `Relay-Server` solution, the `Relay-Server` performs a few more overhead tasks than previously mentioned. Examples of these tasks are the following: starting the benchmark, ending the benchmark, and shutting down processes. All of these tasks are performed either before the benchmark or after the benchmark and will thus not have any bearing on the result.

To ensure idempotency, additional functionality has to be added on top; RPCs do not ensure this property on their own. Therefore, create a list that holds the total amount of RPCs received by a client. Every client receives n RPCs from i clients. The list has to be $(i \cdot n) + 1$ long for this purpose due to zero indexing and the instant lookup functionality we want to achieve, this will be discussed further down. Fill it with zeros, and when a RPC is received, flip the zero of index $((n_{\text{sending}} - 1) \cdot i_{\text{total}}) + i_{\text{sending}}$ to a 1. This indicates that this specific RPC already has been received by keeping track of previously received RPCs. One RPC will have a specific identifier attached to it and will not applied more than once. With this implementation, we do not have to iterate through the list, and the look up will be of $O(1)$.

The communication flow of the `Relay-Server` with CRDTs can be seen in Figure 4.5. In this solution each client sends an RPC at every network tick. Before sending an outgoing RPC, each client instantly transforms the sphere on their end—inducing zero SVL. It notes the `Start-Time`, `End-Time`, and sends a ServerRPC to the server. The server receives the RPC and forwards a ClientRPC to all clients. All clients, including the original sender, will receive

this ClientRPC. However, the original client simply drops the packet without performing another transform. These clients performing the transformation will proceed to note the `End-Time` for that operation and write that to a list.

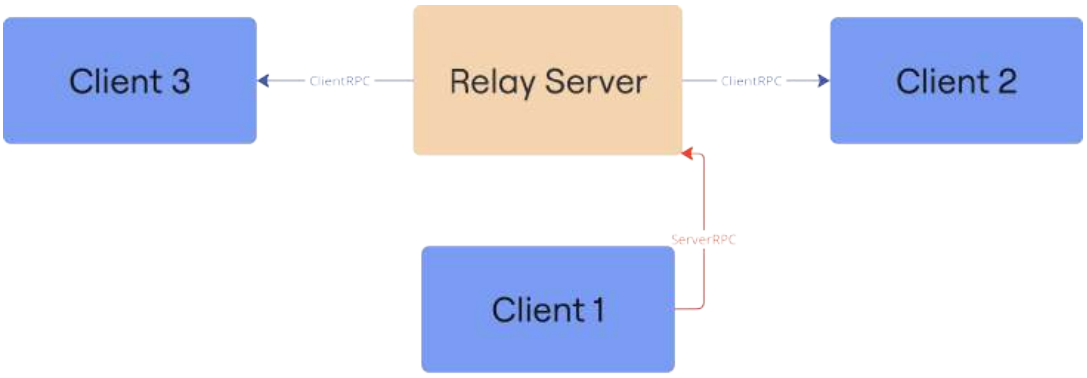

Figure 4.5: Relay-Server RPC Workflow

The reason the ClientRPC is sent back to the original sender is because of Unity's network optimization. The `ObserverList` is used by the RPC to determine the recipients. If we instead opted to exclude a client from the `ObserverList` this incurs runtime logic added to the RPCs. The difference between compile time and runtime RPCs is significant in terms of latency. Therefore, it would be more appropriate to stick to compile time solutions for all topologies. The packet is simply dropped at the original sender's end without modifying the `ObserverList`.

Chapter 5

Results and Analysis

This chapter consists of the results and an analysis of these results. Section 5.1 contains the raw results of the benchmark; the complete results can be found in Appendix A and Appendix B. The hardware setup is explained in Section 5.2. A reliability analysis is performed in Section 5.3, a validity analysis is performed in Section 5.4. Discussion of the results is conducted in Section 5.5. An analysis of the potential gain of introducing CmRDTs with a host is discussed in Section 5.6.

5.1 Major results

In this section, the raw results of the benchmark are shown and discussed. Several iterations of the benchmark were run. As mentioned in Section 3.3 two values were changed during the benchmark, namely the number of clients, where $n \in [2, 16]$, and the amount of iterations $i = \{500, 1000, 1500, 2000\}$. The full results can be seen as scatter plots in Appendix A and B. The evaluation of the results in this chapter will begin with an analysis of the scatter plots. Furthermore, a holistic summary of the scatter plots will be provided, combining the average latency for one test. Lastly, an analysis of the data depicted as violin plots will be presented. With that said, let us start with the scatter plots.

5.1.1 Analysis of Scatter Plots

Scatter plots were created to show the difference in latency throughout one iteration of the benchmark for all topologies. These figures capture the total latency of all transactions, and not only SVL. In Figure 5.1 and Figure 5.2

the `X-Axis` represents the time since the start of the benchmark for each operation, 0 indicates the start of the experiment and the `Start-Time` is converted to ms. The `Y-Axis` represents the `End-Time - Start-Time` i.e., Δt for a given transaction. This value can also be thought of as the total latency of a transaction. When a transaction is initiated by one client, it is received by n clients, including the initiating client. In other words, one transaction results in n amount of Δt. Therefore, one `Start-Time` results in n `End-Times`, and thus yields n dots in Figure 5.1 and Figure 5.2 for that `Start-Time`, or transaction.

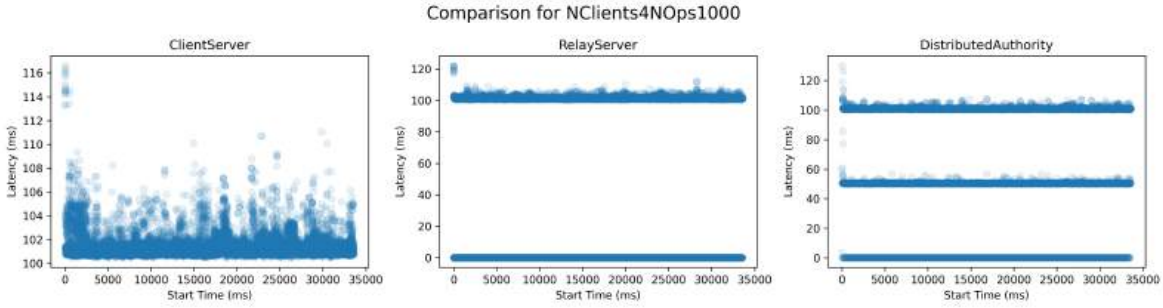

Figure 5.1: Benchmark Results for N=4 Clients and i=1000 operations

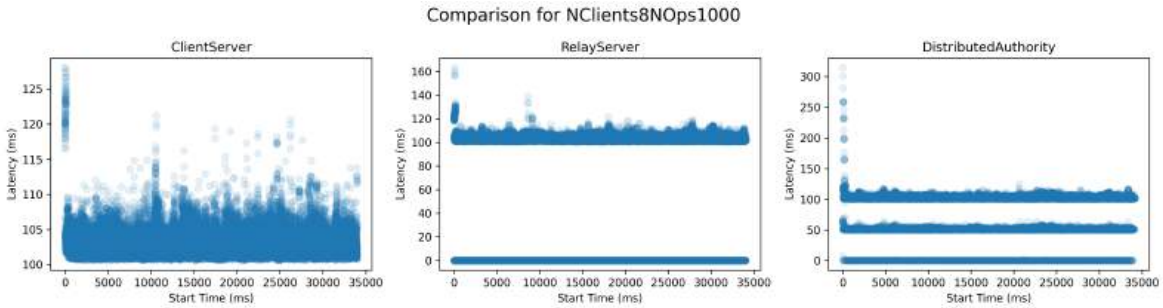

Figure 5.2: Benchmark Results for N=8 Clients and i=1000 operations

If we consider Figure 5.1, the three topologies with the same number of clients, and amount of iterations are displayed as parts of the same figure. Partitions of the Δts for the different topologies can be observed in the graphs of Figure 5.1. These partitions will be covered and explained further by performing a validity analysis in Section 5.4. `Client-Server` has one partition, `Relay-Server` has two partitions, and `Distributed Authority` has three partitions. The added latency for the benchmark was $50ms$. The partition in `Relay-Server` is separated by $2 \cdot 50ms$. The

two partitions in `Distributed Authority` are each separated by $50ms$. In `Client-Server` one partition can be observed. The latency is strictly above $100ms$ for every transaction. If the results of Figure 5.1 are compared with Figure 5.2, the same partition is observed in these figures. This is also true when comparing Figure 5.1 with the graphs in Appendix A and Appendix B. Evidently, the partitions obey the `Network Latency` induced in the experiment as multiples of $50ms$.

In Figure 5.1 and Figure 5.2, both the `Relay-Server` which uses CmRDTs and the `Distributed Authority` show partitions with a close to $0ms$ delay. This indicates that the operations resulting in these Δt values were not sent over the network, as they would suffer a $50ms$ delay every time they passed over the network. Another partition can be seen in the `Distributed Authority`, which hovers just above $50ms$. This partition indicates that the operations resulting in these Δt values were sent over the network once, suffering a $50ms$ delay. To further facilitate a discussion into these partitions, we now let the partition close to $0ms$ that was not transferred over the network be denoted as 0 RTT. We denote the partition just above $50ms$, that was transferred over the network once, as 0.5 RTT. Lastly, the partition just above $100ms$, that was transferred over the network twice, is denoted as 1 RTT.

A *Warm-Up Time* is observed in Figures 5.1 and Figure 5.2; this *Warm-Up Time* is especially apparent in Figure 5.2, although the *Warm-Up Time* is present in all iterations of the benchmark. These larger Δts occur at the start of the benchmark. This can be viewed as the `Overhead Latency` being very high during the start of the benchmark. For example, in the `Client-Server` part of Figure 5.2, the effect of *Warm-Up Time*, as added latency, is visible as a discrete line of dots of high y-values close to $x = 0$.

5.1.2 Analysis When Combining The Scatter Plots

As mentioned in the previous section, all topologies have different partitions. If we want to analyze the latency with respect to a specific partition, we can consider the partition that is native to all three topologies; the partition of values just above $100ms$ latency, or the 1 RTT partition.

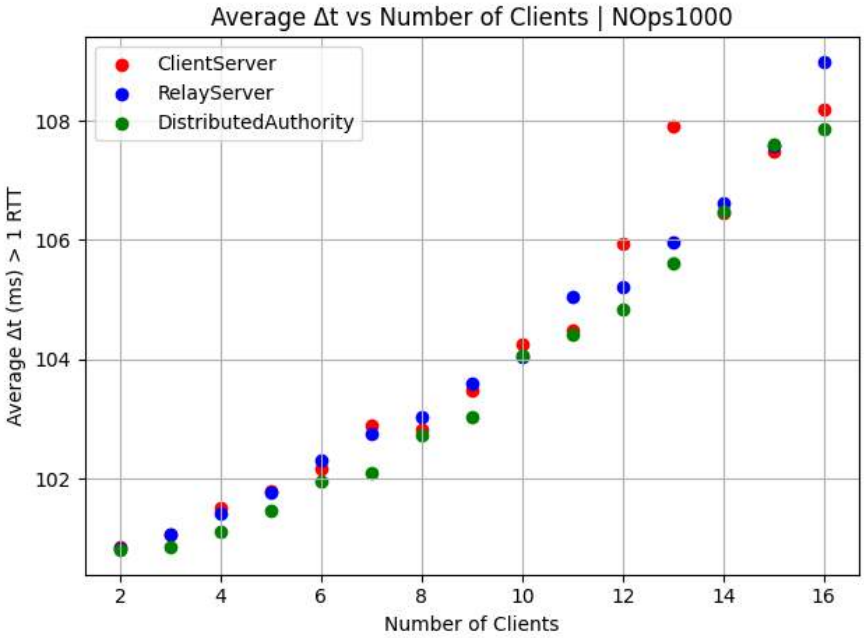

Figure 5.3: Average Delta t when increasing clients | i = 1000

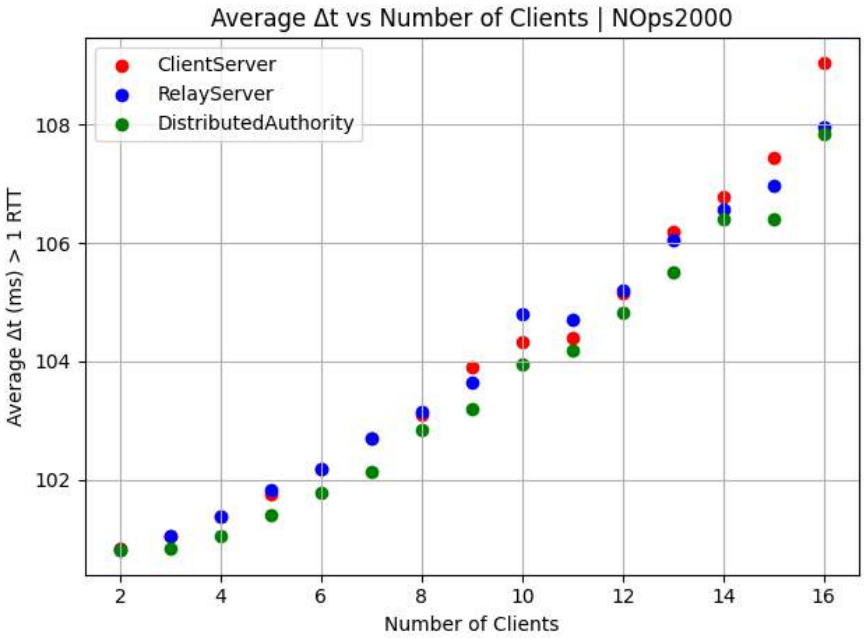

Figure 5.4: Average Delta t when increasing clients | i = 2000

The average latency of the 1 RTT partition can be seen in Figure 5.3 and Figure 5.4. In these figures, the `X-Axis` shows the number of clients, and the `Y-Axis` shows the average Δt for all transformations that took more than $100ms$ to complete. These figures show an increase in `Overhead Latency` as the number of clients increases. The purpose of this depiction is to validate the data and differentiate a trend in the data. `Client-Server` is characterized by the red dots, `Relay-Server` is characterized by the blue dots, and `Distributed Authority` is characterized by the green dots. What becomes apparent in Figure 5.3 and Figure 5.4, is that, when the number of clients increase, the average Δt increases. In other words, an iteration of the benchmark with a larger number of clients incurs an increased `Overhead Latency`.

5.1.3 Analysis of Violin Plots

To differentiate the number of transactions belonging to each partition, violin plots were created. Two of these violin plots, based on two different iterations of the benchmark, can be seen in Figure 5.5 and Figure 5.6; the figures show the latency distribution for each topology. Additionally, the figures show the same partitions as the scatter plots seen in Figure 5.1 and Figure 5.2. Furthermore, Figure 5.5 and Figure 5.6 show how the latency is distributed between these partitions for one iteration of the benchmark.

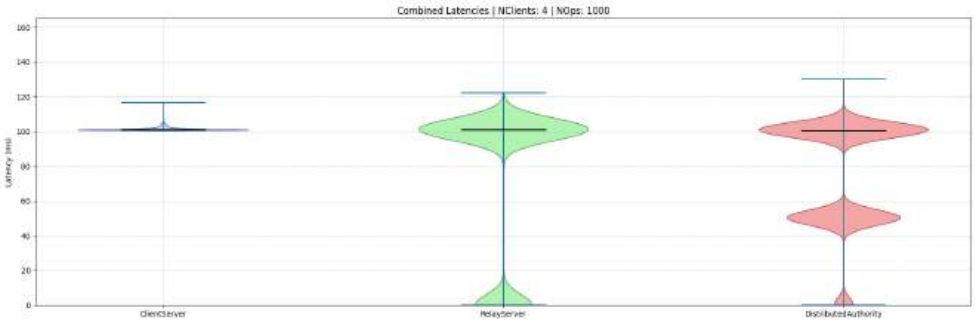

Figure 5.5: Violin Plot: All Topologies | n = 4, i = 1000

Interestingly, when comparing Figure 5.5 and Figure 5.6, it becomes evident that the number of transactions belonging to the $0ms$ partition and the $50ms$ partition decrease as the number of clients increase. As noted in Section 5.1.1, our plots, now also including the violin plots, highlight the total latency of any given transaction. However, we are mostly interested in the latencies pertaining to SVL, therefore the task of identifying which latencies

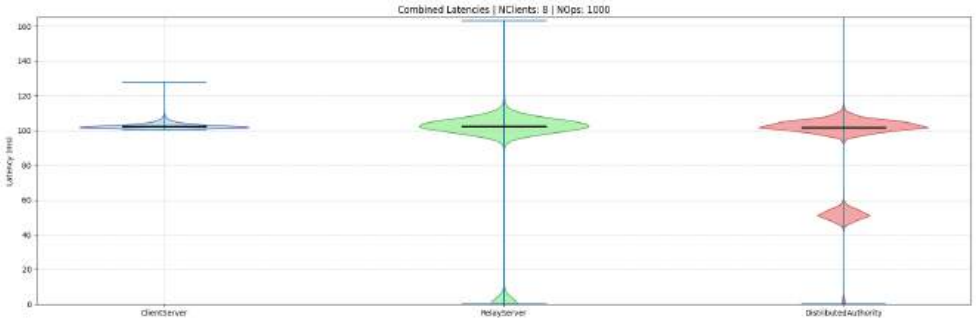

Figure 5.6: Violin Plot: All Topologies | n = 8, i = 1000

belong to this category becomes pertinent. In other words, which latencies in these figures do actually represent SVL? To clearly differentiate this, the SVL can be separated from other latency and be displayed as additional violin plots. By isolating the `End-Time` belonging to the client that issues an transaction, from the `End-Times` of all other clients, we can differentiate between the values that pertain to SVL.

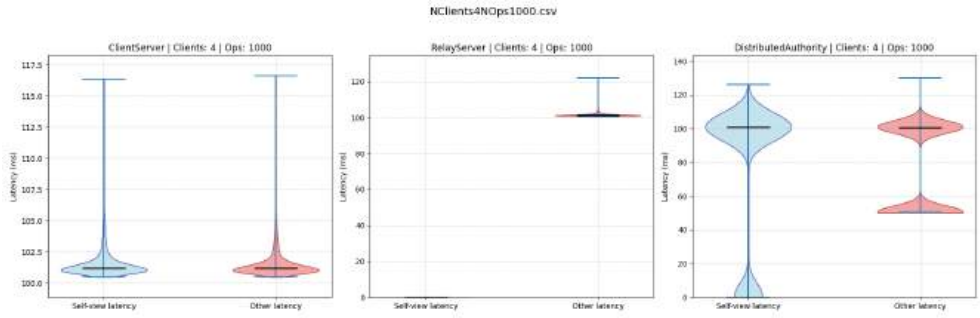

Figure 5.7: Violin Plot: All Topologies | Separated SVL | n = 4, i = 1000

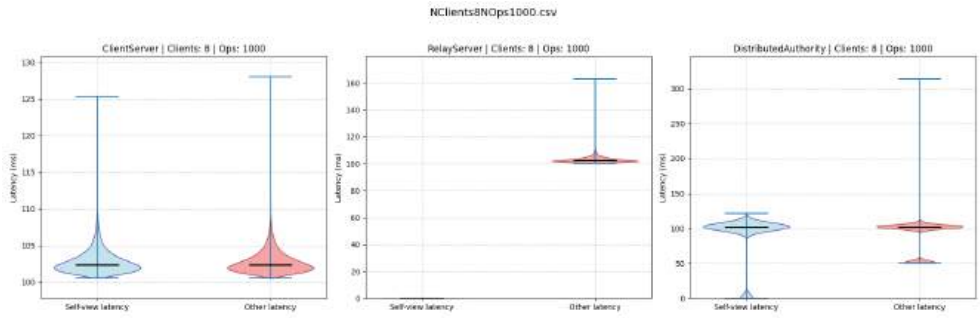

Figure 5.8: Violin Plot: All Topologies | Separated SVL | n = 8, i = 1000

The results of the violin plots with isolated data points for SVL can be seen in Figure 5.7 and Figure 5.8. When looking at these figures, the SVL sections and the its correspondence show the same width, indicating that their cardinalities are similar. This is somewhat deceiving as their cardinalities vary dramatically, and is an issue relating to the plotting mechanism itself. As a reminder, as mentioned in Section 3.3, the total number of RPCs sent is $i \cdot n$ and received is $i \cdot n \cdot n$. Therefore, the total number of Δts pertaining to SVL is $i \cdot n$. Furthermore, the total number of Δts pertaining to other latency not attributable to SVL is $i \cdot n \cdot n - i \cdot n = i \cdot n \cdot (n-1)$.

Figure 5.7 and Figure 5.8 show that all topologies have different partitions of SVL. Let us start with the `Client-Server` topology; in the `Client-Server` topology the SVL is very similar to the latency experienced by other clients. All data points in this topology belong to the partition of $100ms$ since all traffic has to be routed through the server before any actions can take place. Looking at the `Relay-Server` with CmRDTs, the entire 0 RTT partition is considered to be SVL, while the 1 RTT partition is considered to be non-SVL, i.e., latency experienced by other clients. In the `Distributed Authority` topology the SVL occurs at both the 0 RTT partition and the 1 RTT partition.

5.2 Hardware Setup

Designing a benchmark program to run a local computer introduces two problems; personal computer systems are inherently limited in their computing power. This minimizes the possibility of extracting more data points, and thus limits the scope of the benchmark. Using a personal laptop to execute the benchmark also posits another issue; additional processes running in the background, and using the laptop during a benchmark could potentially skew the results. Running it on a minimalist cloud server removes both of these issues while also adding more computing power to increase our data points. Moreover, simulating the game across several different systems would also introduce some additional problems. Latency would be drastically different between different machines. The values from the benchmark would be harder to interpret, as we are adding more layers of complexity. Therefore, running the benchmark on one system would be preferable. The benchmark is executed on one system, the KTH cloud server. The cloud server used had 12 Central Processing Unit (CPU) cores and 80 Gigabyte (GB) of Random-access Memory (RAM) memory, and uses Linux.

5.3 Reliability Analysis

The purpose of the reliability analysis is to evaluate whether the results obtained by performing the benchmark are repeatable and consistent. In other words, is the benchmark replicable [64]? This will be done by analyzing the data from three reliability metrics: *Repeatability, Robustness, and Error Sources*.

5.3.1 Repeatability

Repeatability of the results means that similar results are obtained when repeating the experiment. The repeatability of the data was ensured by performing the benchmark a significant amount of times. We varied the number of clients between 2-16 clients and the amount of operations used were: 500, 1000, 1500, and 2000. Therefore, a total of $15 \cdot 4 = 60$ iterations of the benchmark were performed for each of the three topologies. The mean Δt for operations that exceeded $100ms$ latency can be seen in Figure 5.3 and Figure 5.3. The depiction of $100ms$ as the baseline latency was chosen in these figures since all topologies have transformations over $100ms$ but not all of them have transformations below $100ms$. Therefore, it is most meaningful to compare values within the same latency range, which is common for all topologies. These figures show that the mean Δt varies only slightly when comparing the different benchmarks. These consistent results ensure the repeatability of the benchmark.

To ensure the correct amount of Δts are reflected, reliable RPCs were used. Although RPCs use UDP, the reliability function provided by Unity ensures that the RPC reaches every client. Furthermore, to ensure the reliability of the system the benchmarking program only exits with the proper results if the correct amount of `Start-Times` and `End-Times` have been documented. This was added as a fail-safe during testing to ensure that the program did not yield any incomplete tests. Therefore, no data was lost for any of the benchmarks and they all contain complete data.

5.3.2 Robustness

To ensure the robustness of the benchmark, we consider different results of the benchmark when changing a variable. Consider the case when we increase the number of clients. An example of this can be seen in Figure 5.3 and

Figure 5.4. The sum of the Δt values vary only slightly between topologies, and in all these figures the results follow the same trend. The benchmarks were performed with an increasing number of clients; these figures highlight an increase in the Δts of each topology when increasing the number of clients. This phenomenon cannot be explained by the `Network Latency` as it is a fixed value of $2 \cdot 50ms = 100ms$. This means that the increase in `Overhead Latency` must occur as a result of the stress experienced by the server authority as the number of clients increase. When performing the benchmark, as many variables as possible were kept static, background scripts were identical, as well as the underlying code behind the benchmark. The combination of these patterns in the data, as well as the static background variables, yield robustness of the data collected in the benchmark.

5.3.3 Error Sources

To control for sources of error, precautions were taken when designing the benchmark. For example, to ensure reliability of the data, localhost was used for determining a set `Network Latency` of a transaction. The `Network Latency` could be set to $50ms$ to reduce the margin of error that would occur when running on separate systems. In the case of a non-static latency, the individual `Network Latency` for each system would come into effect. These network latencies would be different depending on the distance between the peers. Therefore, the test was run on a cloud server simulating all clients and the server authority. This also allowed us to use a synchronized processor clock to document the time for the individual transactions. As such, the clients are able to individually save all the `Start-Times` and `End-Times` from their system with no discrepancies. These files can later be merged into a large file that contains the combined files of a benchmark. Being able to use the processor clock was very beneficial when tracking extremely small times that were apparent between systems.

As mentioned in Section 5.1, a *Warm-Up Time* of time of the system can be noted on the graphs, which is expected. This can be seen in Figure 5.1 and Figure 5.2 at the beginning of the test, close to 0 on the `X-axis`. Among these data points, some of the latency values were unusually high, and as a result they might occasionally belong to a different partition than they would otherwise, as in the case of `Distributed Authority`. More specifically, operations transferred over the network once, that normally only would suffer a $50ms$ delay, may be shown to have a latency over $100ms$ instead due to *Warm-Up*.

In this thesis, we let a $50ms$ delay be considered as a transaction with 0.5 RTT, and a $100ms$ delay as 1 RTT. For this reason, inducing a delay with a value of $50ms$ now becomes clearer, as we would like to section them far apart for the purpose of visualizing what happens as the different partitions are separated. For a latency analysis the values belonging to *Warm-Up Time* would be completely disregarded, in fact the complete beginning of the test could be disregarded in this case. This is because in a interactive application session we are not interested in what happens within the first few ms of a session, but rather throughout the session as a whole.

Lastly, to ensure *idempotency* of the CmRDT in the case of the `Relay-Server`, additional measurements were taken to ensure that transactions with the same identifier are only applied once. By introducing these measurements, combined with Unity's reliability functionality of the RPCs, we ensure that all data will arrive once and only once for a given transaction. Additionally, we ensure that a benchmark can only finish with the correct amount of RPCs received. This removes potential error sources where `End-Times` could be overwritten due to repeated RPCs.

5.4 Validity Analysis

The purpose of this analysis is to see if the results obtained are valid. This is done by examining different types of validity. Internal validity, External validity, and Construct validity will be examined for the benchmark and the collected data.

5.4.1 Internal Validity

To ensure that the results are correct with respect to our method and not influenced by experimental error, or other background factors, we evaluate the *internal validity* of this benchmark [64]. To ensure *internal validity* of the collected data, considerations had to be taken when creating the benchmark environment. All topologies were made to be as similar as possible except for the transform protocols that are inherently different for each of the topologies. Initiating lists, writing to files, and waiting for clients to connect were all performed outside of the benchmark. The same method for taking times, and storing times, were also consistent between topologies. The same hardware was used for all three topologies, and we tried to minimize unforeseen background variables, such as processes changing throughout the benchmark,

that could possibly interfere. This was solved by running the benchmark on a cloud server. On the cloud server, all iterations of the benchmark were run without any external interaction or additional input to the virtual machine.

There are a few outliers visible in Figure 5.3 and Figure 5.4, although the majority of the plot points all follow a trend where the average Δt increases as we increase the number of clients. Furthermore, data consistency can also be observed in Figure 5.1 and Figure 5.2. They all follow the same trend, and with the data for one partition sectioned around a straight line, there is some noise visible. The noise increases as we increase the number of clients; however, this noise is consistent across all topologies. The sections visible in Figure 5.1 and Figure 5.2 are all partitioned according to multiples of the induced `Network Latency` of $50ms$.

The observations of the trends in Figure 5.3 and Figure 5.4, combined with the consistency in Figure 5.1 and Figure 5.2, support the internal validity of the data collected by performing this benchmark.

5.4.2 External Validity

To analyze the external validity of the results of the benchmark, we have to consider whether the results could be generalized outside of the simulated environment in which we obtained them [64]. In other words, external validity is concerned with if the results obtained in this environment can be applicable to other environments, not inherent to this benchmark.

What could be difficult to motivate in a non-test environment, is if the actual `Overhead Latency` is affected positively or negatively when introducing CmRDTs to NGO; when considering the `Overhead Latency` in Figure 5.3 and Figure 5.4 they follow a specific trend, and our results did not provide a definite distinction. Although, in 52 out of 60 iterations of the benchmark, `Distributed Authority` had the lowest latency. This phenomenon could likely be accredited to the fact the `Distributed Authority` topology uses a host instead of a server as a server authority. The host acts a both a client and a host. As a consequence, this means that whenever the host receives an RPC from a client to update all other clients, it can simply locally transform its own update. This results in one less RPC being sent over the network, and could have implications for the overall `Overhead Latency`. The results would have to be repeated many more times to be

able to definitively impose a definite trend. The test environment would have to be further adjusted, testing a host solution with CmRDTs to validate that this is the reason for this reduction. Therefore, we are unable to draw any definite conclusions concerning the change in `Overhead Latency` via the introduction of CmRDTs.

By implementing CmRDTs in Unity with Unity's own networking library NGO, we show that including CmRDTs in existing game engines is a real possibility; that this is not restricted to theory but also applicable in practice. By showing that this type of implementation is possible in practice, this supports the external validity of this thesis and that this can be generalized outside the scope of this environment. Furthermore, by modifying existing network topologies in NGO, with a varying player amount to simulate real-world scenarios for testing, this further supports the external validity of this thesis. However, it should be noted that this implementation has limitations concerning the nature of the `Relay-Server` and `Distributed Authority` implementation. As such, it is not clear whether the results of the benchmark would appear similar on a `Distributed Authority` solution provided by Unity's cloud services. Furthermore, since the benchmark is constrained to one processor, it is unclear how the `Overhead Latency` would behave in a real distributed environment.

5.4.3 Construct Validity

We perform an analysis of construct validity to validate that the data collected actually represent what we are interested in. In other words, if the introduction of CmRDTs in existing Unity network topologies actually yield a zero SVL, or if the difference in latency is inherent to some other factor. By analyzing the mathematical distribution of the Δts we can gain insight in what types of operations yield a zero SVL. This is interesting because we can see that the `Distributed Authority` and the `Relay-Server` both have partitions. These partitions can be seen in Figure 5.5 and Figure 5.6. A mathematical comparison can yield additional information about the *total* RTT of one benchmark. This could yield insight in how much network overhead could be reduced by introducing CmRDTs in current Unity networking frameworks. Furthermore, this can verify indeed that the correct amount of RPCs are received for the correct values.

5.4.3.1 A Mathematical Analysis for The RTT Distribution

This section aims to perform a mathematical analysis for the purpose of corroborating the construct validity of the results. To begin, for a given client the total number of messages sent will be the number of operations i. Each client sends i messages, therefore a total of $i \cdot n$ messages are sent by clients for a given benchmark. Each client receives the corresponding message sent by the sending client. Therefore, the total number of Δts results in $i \cdot n \cdot n$.

The structure of the analysis, will be conducted as follows. For each topology, we analyze the sender, receiver, and RTT of the RPC. We then derive a formula that describes the cardinality of each RPC belonging to a certain RTT category. The RTT catogories for this analysis will be 0 RTT, 0.5 RTT, and 1 RTT. By categorizing these values as discrete values we hope to make the analysis of what latency pertains to zero SVL more apparent.

To evaluate the correctness of our benchmarks and their measured latencies, we expect that all topologies, despite their partitions, result in a total of $i \cdot n \cdot n$ operations. This fact will be used to verify the correctness of each formula.

Client-Server

In the `Client-Server` topology there is only one type of transaction. This type of transaction can be seen in Figure 5.9.

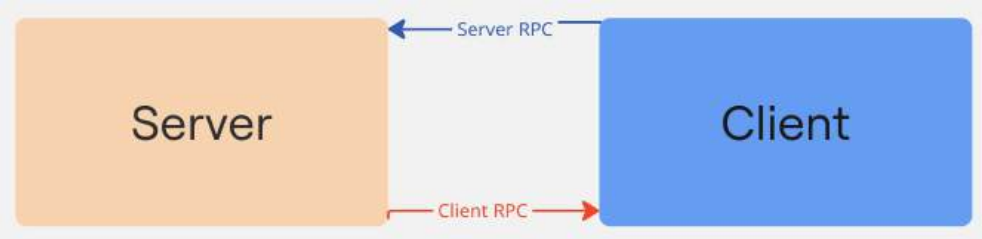

Figure 5.9: Client-Server RTT graph | Client-to-Client

In Figure 5.9, when a client communicates with another client, the outgoing client sends a ServerRPC to the server with a 0.5 RTT. Since all communication between clients has to go through the server, the minimum, and maximum RTT for this topology is 1 RTT. As previously mentioned, the the total number of Δts amount to $i \cdot n \cdot n$. In the case of `Client-Server`, the distribution of RTT can be seen in Figure 5.10.

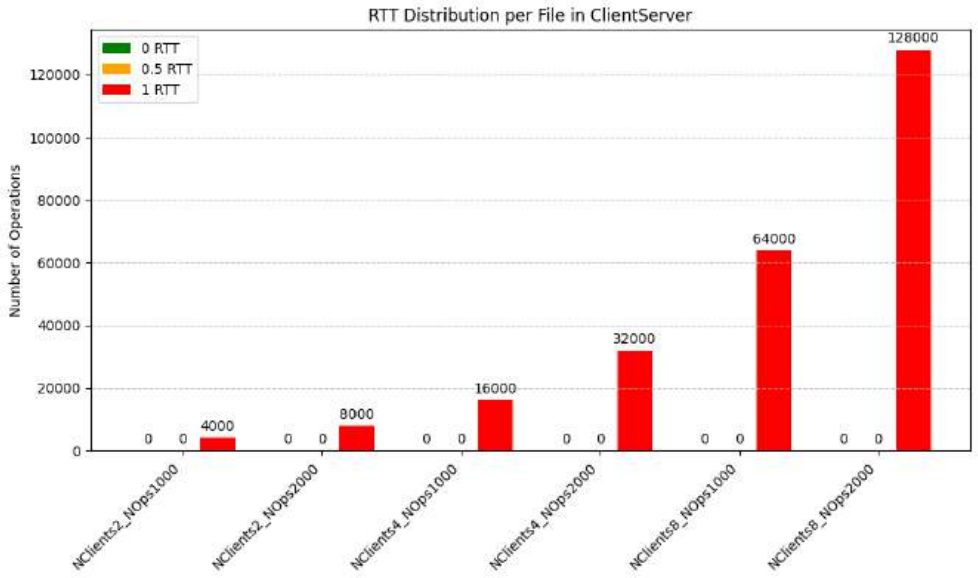

Figure 5.10: Client-Server RTT Bucket Graph

When comparing the total number of RTTs with the RTTs in Figure 5.10, it is observed that $i \cdot n \cdot n$ messages are received by clients. Additionally, that all of the $i \cdot n \cdot n$ messages arrive with an RTT of 1. For example, consider the case of $N = 2$, $i = 1000$. As per $i \cdot n \cdot n$, the number of RTTs in this case would be $1000 \cdot 2 \cdot 2 = 4000$. Looking at the graph, we see that 4000 transactions are observed, all in the 1 RTT bucket.

Custom Distributed Authority

In the simulated `Distributed Authority` topology there are four different types of transactions. We now aim to categorize these and show that the amount of Δts, as in the case of the two previous topologies, also amounts to $i \cdot n \cdot n$. These transactions can be seen in Figure 5.11, Figure 5.12, Figure 5.13, and Figure 5.14.

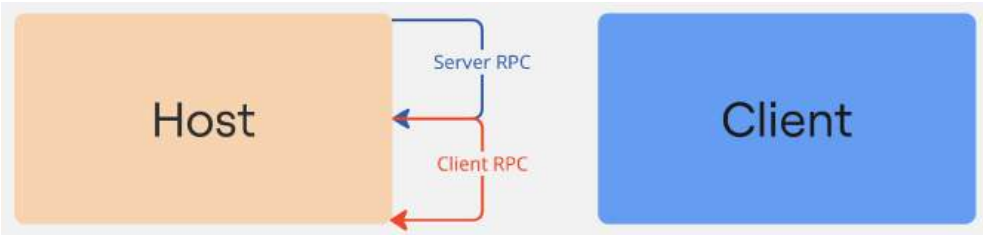

Figure 5.11: Distributed Authority RTT graph | Host-to-Host

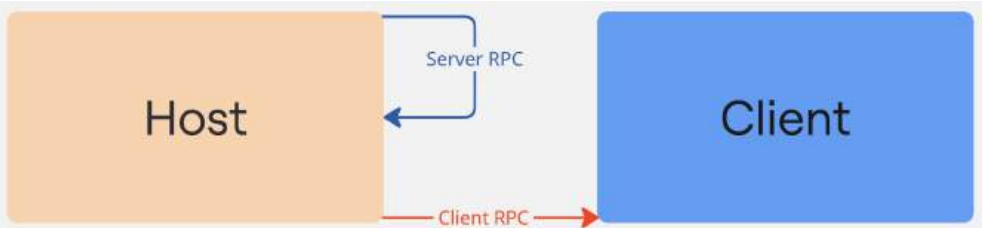

Figure 5.12: Distributed Authority RTT graph | Host-to-Client

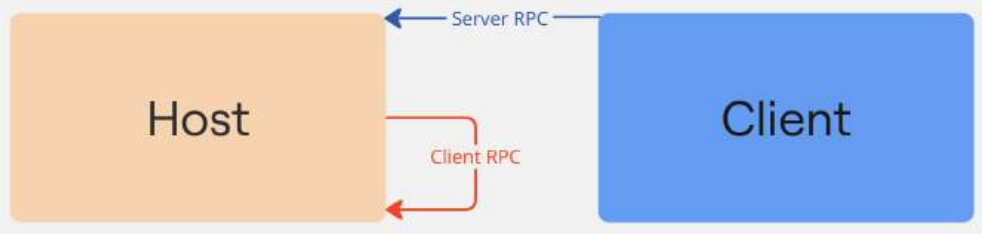

Figure 5.13: Distributed Authority RTT graph | Client-to-Host

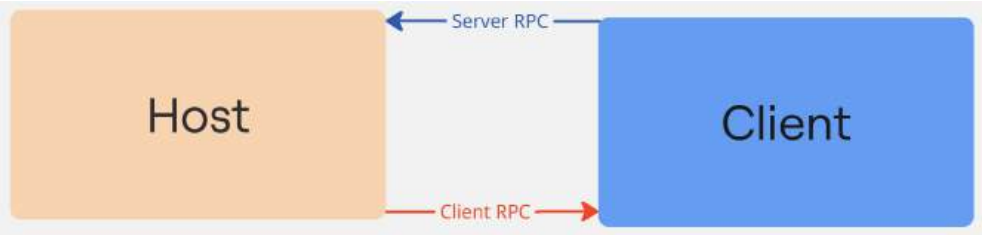

Figure 5.14: Distributed Authority RTT graph | Client-to-Client

The first case can be seen in Figure 5.11. In this case the host is the sender and receiver of the RPC and the RTT results in 0. The amount of Δts that this category results in is i. This is because the host sends i messages and the host is the single receiver of these RPCs in this case.

The second case can be seen in Figure 5.12. In this case the host sends an RPC. The server RPC is sent internally to the host and then broadcasted to all clients. The host sends i messages and this is received by $n-1$ clients. Therefore, this results in $i \cdot (n-1)$ total number of Δts.

The third case can be seen in Figure 5.13. In this case the client sends a server RPC to the host, the host then proceeds to transform internally, and the host is the end point for this transaction. There are $n-1$ clients, sending i operations,

being received once. Therefore, this results in $(n-1) \cdot i$ number of Δts.

In the fourth case, which can be seen in Figure 5.14 the client sends an RPC and the receiver is also a client. The amount of RPCs sent is $i \cdot (n-1)$ and this is received by $(n-1)$ clients. This results in $i \cdot (n-1)(n-1)$ total number of Δts.

If we add all cases together the following result is obtained: $i, i(n-1), i(n-1)$, and $i(n-1)(n-1)$. This results in, $i + 2i(n-1) + i(n-1)(n-1)$ = $i(1 + 2n - 2 + n^2 - 2n + 1)$ = i(n^2) = $i \cdot n \cdot n$. Therefore, this results in the expected amount of Δts as initially suggested. A comparison with our results can be seen in Figure 5.15. Some outliers are visible in these buckets due to some transactions of 0.5 RTT being documented as 1 RTT. This is due to the *Warm-Up Time* of the system, where transactions take longer, and they occur in the first few transactions. To validate this the values that resulted in 0, or 0.5 RTT were separated. These values were plotted to verify that they occurred at the very start of the benchmark. An example of this can be seen in Figure 5.16, and verified with the values in Figure 5.15. In Figure 5.16 two values exceed $100ms$ and that corresponds to the `NClients4NOps2000` section of Figure 5.15.

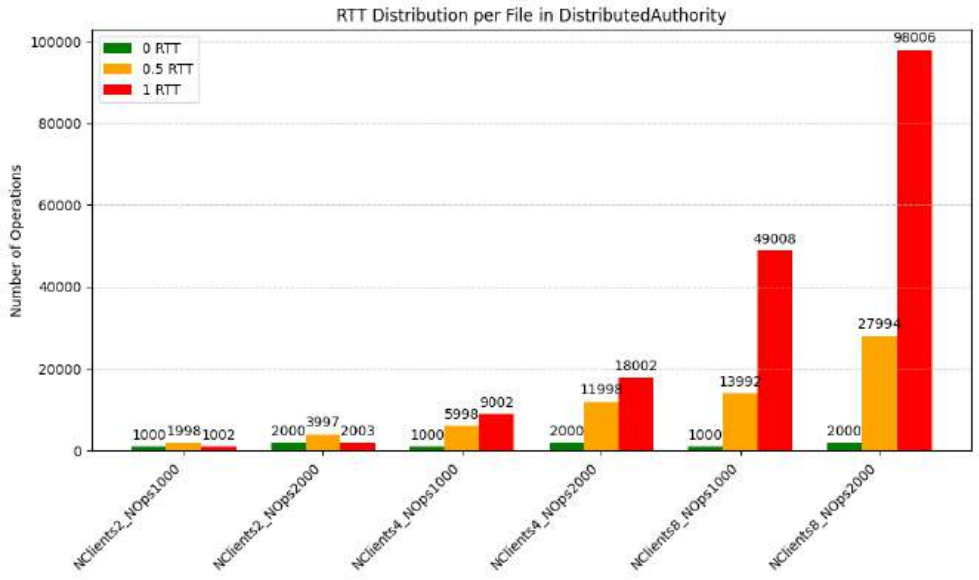

Figure 5.15: Distributed Authority RTT Bucket Graph

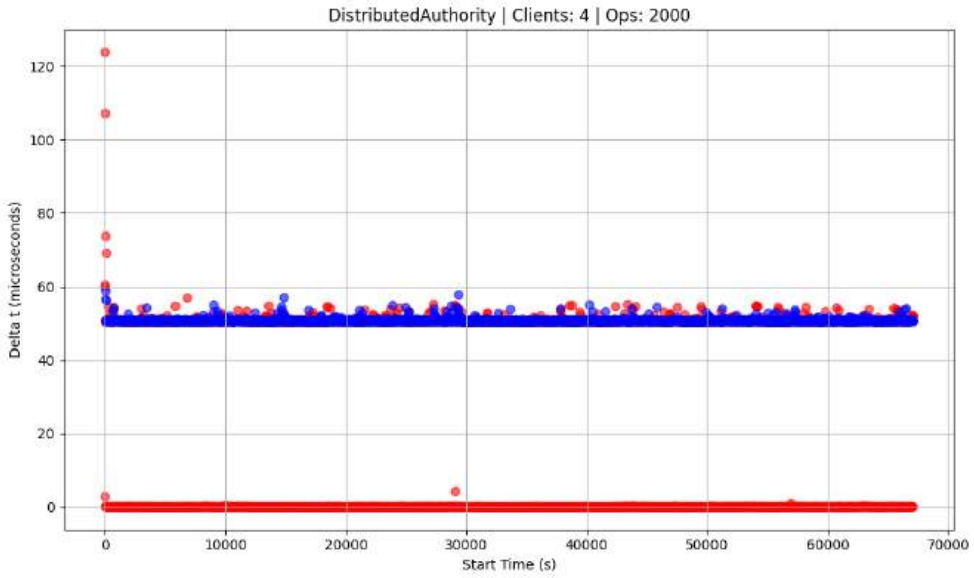

Figure 5.16: Overflow of 0.5 RTT Values in Distributed Authority

Relay-Server with CmRDTs

In `Relay-Server` there are two different types of possible transactions. These transactions can be seen in Figure 5.17 and Figure 5.18.

Figure 5.17: Relay-Server RTT graph | Client-to-Self

Figure 5.18: Relay-Server RTT graph | Client-to-Other

To formulate these cardinalities, the two figures highlight two different types of RPC transactions within this topology. Figure 5.17 depicts the first case. In the first case, the client simply performs a local-first update, as part of

the *effect-update* of the CmRDT implementation. The RTT of this case results in 0. The amount of transactions resulting in this category are as follows: For each client's operation, a local-first update is performed before sharing it across clients. This instance occurs for every client's operation, or, outgoing RPC. This results in $i \cdot n$ operations that fall under a 0 RTT category.

The second case in Figure 5.18, depicts the sharing of local updates between two clients. In this case, all communication between clients must go through the `Relay-Server` resulting in a total RTT of 1. The number of received RPCs in this category is the result of i operations being sent by n clients and received by $n-1$ clients. Therefore, the resulting number of Δts amounts to $i \cdot n \cdot (n-1)$.

With these two cases, the total number of Δts should still be equal to $i \cdot n \cdot n$. The first and second case added together give us the equation:
$(i \cdot n) + (n-1) \cdot (i \cdot n) = (i \cdot n \cdot n)$ The left hand expression can be simplified as follows: $(i \cdot n) + (n-1) \cdot (i \cdot n) = (i \cdot n)(1 + (n-1)) = i \cdot n \cdot n$

Figure 5.19, shows the sum of the different RTT buckets obtained from the benchmark. The mathematical analysis of this section supports these results. Furthermore, Figure 5.6 from Section 5.1 showed us, for the `Relay-Server` topology, that all operations resulting in 0 RTT pertains to SVL and all operations resulting in 1 RTT pertains to non-SVL. Therefore, the green staples seen in Figure 5.19 all pertains to SVL; meanwhile, the red staples pertains to non-SVL.

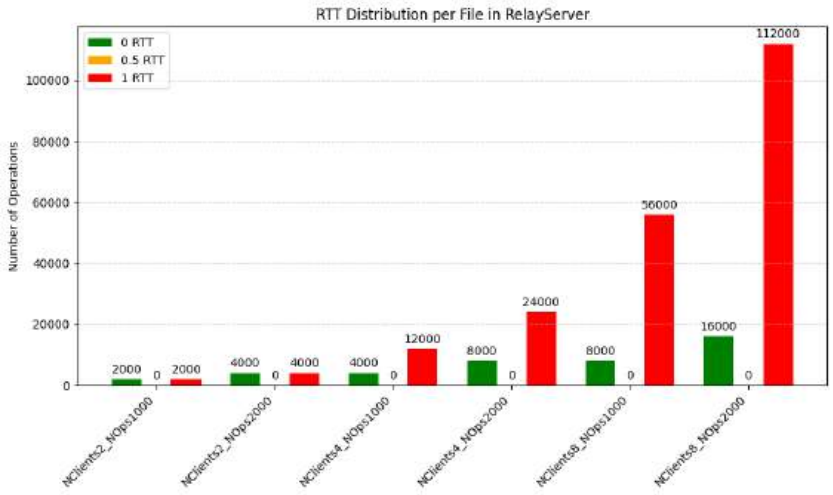

Figure 5.19: Relay-Server RTT Bucket Graph

5.5 A Closer Look at The Different RTTs

The difference in overall RTT distribution between the different topologies can be seen through the different mathematical formulae in Table 5.1. We can see that `Client-Server` only has Δts that result in 1 RTT. `Relay-Server` has both Δts that result in 0 RTT, and 1 RTT. `Distributed Authority` has Δts that result in 0 RTT, 0.5 RTT, and 1 RTT. The frequencies at which they occur at are visible as formulae where the number of clients are noted as n, and the amount of operations are noted as i. The total amount of Δts are $i \cdot n \cdot n$. Furthermore, the results obtained by this mathematical analysis help ensure the validity of the data, and was discussed in Chapter 3, specifically Section 3.5.3. The results of the mathematical analysis are compiled in Table 5.1.

Table 5.1: Mathematical Formulae for RTT Distribution in the Different Topologies, RS - Relay-Server, DA - Distributed Authority

	Client-Server	RS with CRDT	Custom DA
0 RTT	0	$i \cdot n$	i
0.5 RTT	0	0	$2 \cdot i \cdot (n-1)$
1 RTT	$i \cdot n \cdot n$	$i \cdot n \cdot (n-1)$	$i \cdot (n-1)(n-1)$

The reduction of the total RTT experienced during a benchmark with `Relay-Server` and `Distributed Authority` can thus be inferred as follows. If we multiply the formulae from Table 5.1 with their corresponding RTTs we get the total reduced RTT that each different topologies yield.

For `Client-Server` the result is:

$$1 \cdot (i \cdot n \cdot n) = i \cdot n \cdot n$$

For the `Relay-Server` solution the result is:

$$0 \cdot (i \cdot n) + 1(i \cdot n \cdot (n-1)) = i \cdot n \cdot (n-1)$$

For the `Distributed Authority` the result is:

$$0 \cdot (i) + 0.5 \cdot (2i(n-1)) + 1 \cdot (i \cdot (n-1)(n-1)) =$$
$$i \cdot ((n-1) + (n-1)(n-1)) = i \cdot (n^2 - n) = i \cdot n \cdot (n-1)$$

Therefore, we see that the total contributed RTT by operations of the `Relay-Server` using CmRDTs and the `Distributed Authority` solution yield the same reduction in RTT. These combined results can be seen in Table 5.2.

Table 5.2: The total amount of RTT experienced by the different topologies

	Total RTT Per Benchmark
Client-Server	$i \cdot n \cdot n$
Relay-Server with CRDT	$i \cdot n \cdot (n-1)$
Custom Distributed Authority	$i \cdot n \cdot (n-1)$

A pertinent question arises; how is this experienced by the user? For the `Distributed Authority` solution only the host experiences the 0 RTT scenarios when it locally performs the updates. Therefore only one of the clients, the host, experiences these benefits. Furthermore, for the 0.5 RTT, the reduction in latency only applies to the transactions from the host, or to the host. Therefore, the host experience this type of beneficial latency reduction with an RTT of 0. The clients experience some reduction when the host is the node performing an operation on the sphere object. This occurs with a frequency of $i(n-1)/inn$ in our benchmark.

If we instead consider the CmRDT solution, the reduction in RTT benefits all clients equally. Whenever a client interacts with the sphere, it is known that the update of the sphere eventually will converge across all clients. Therefore, as a client instantly updates the position of the sphere, it experiences zero `Network Latency`, or zero SVL. However, all other clients have not yet received this update, and as such will need to be transferred over the network, inducing RTT delay. The important implication here is that updates performed

on an object will be perceived as having zero latency by the actor, and observers of that object will receive this update at a later date.

A high SVL is especially noticeable when a user's own actions are delayed. This thesis aims to reduce this specific form of latency. Other clients actions are still delayed. A user's own actions follow a local first paradigm, by the use of CmRDTs, this results in a low SVL. As mentioned in the introduction, this becomes particularly important in industries such as XR, where a user's senses are more sensitive to the delay of their own actions.

5.6 Potential Benefit of Using a Client-Host With CmRDTs

The `Distributed Authority` solution uses a host, while the `Relay-Server` uses a server. In the context of `Distributed Authority`, one benefit of reduced RTT is credited to the host being able to perform local updates and experience minimal latency when routing through itself. Consider combining a host with the CmRDT solution instead of a server. In such a case, further reduction to the total RTT would be achieved. With the knowledge gathered of the RPC workflow of Unity's NGO we can anticipate this reduction as follows.

5.6.1 RTT Under a Client-Host Solution Leveraging CmRDTs

The total amount of 0 RTT transformations remains $i \cdot n$; these transformations occur only when a client updates locally for every operation, in other words once for each outgoing RPC.

The total amount of 0.5 RTT transformations will be, as in the case of `Distributed Authority`, $i \cdot (n-1)$ for every outgoing RPC sent by the host. This is because this event occurs i times and will be received by $(n-1)$ clients. Similarly, as a client initiates an update and sends it to the host, this event occurs i times, and will be sent by $(n-1)$ clients. This event is only received at one node, the host, and becomes a constant of 1. In total, this occurs $i \cdot (n-1)$ times. Therefore, the total frequency of 0.5 RTT transformations is $2 \cdot i \cdot (n-1)$.

The 1 RTT transformations occur when a client sends an update, and the

recipient is neither the initial client nor the host. In other words, the only nodes that would receive an operation that results in 1 RTT would be clients, except the original sending client. This event will result in $i \cdot (n-1)(n-2)$ clients receiving the update. This is due to i transformations performed by $(n-1)$ clients and received by $(n-2)$ clients. To elaborate on the $(n-2)$ factor, it is credited to both the initial client and the host. The initial client being a recipient would result in a 0 RTT operation, and the host being a recipient would result in a 0.5 RTT operation. In summary, the formula for the total RTT for a host implementation of the CmRDT solution would then be:

$$
\begin{gathered}
0 \cdot (i \cdot n) + 0.5 \cdot (2i(n-1)) + 1 \cdot (i \cdot (n-1)(n-2)) = \\
i \cdot ((n-1) + (n-1)(n-2)) = \\
i \cdot (n - 1 + n^2 - 3n + 2) = \\
i \cdot (n^2 - 2n + 1) = \\
i \cdot ((n-1)^2)
\end{gathered}
$$

The results from this mathematical analysis can be seen in Table 5.3.

Table 5.3: The different RTT buckets when using a Host with CmRDTs

	Formula
0 RTT	$i \cdot n$
0.5 RTT	$2 \cdot i \cdot (n-1)$
1 RTT	$i \cdot (n-1)(n-2)$
Total RTT	$i \cdot ((n-1)^2)$

5.6.2 An Observed Benefit

The difference between using a host and a server as authority can be seen in Figure 5.20. The total RTT of the different topologies can be seen with their respective formula from Table 5.2 and Table 5.3. If the RTT were to be evaluated on this topology with 16 clients and 2000 operations, which represents highest values used during the benchmark; this amounts to a total RTT reduction of 62000 when compared to `Client-Server`, and 30000 when compared to the `Relay-Server` using CmRDTs. The main advantage of this reduction of introducing a host and evaluated from a SVL perspective, is putting less stress on the network since this reduction of RTT is vaguely experienced by the clients and more experienced by the host. As the introduction of a host only leads to a reduction in SVL for RPCs that start or end with the host, and these RPCs have a 0.5 RTT, these transformations do not result in a zero SVL.

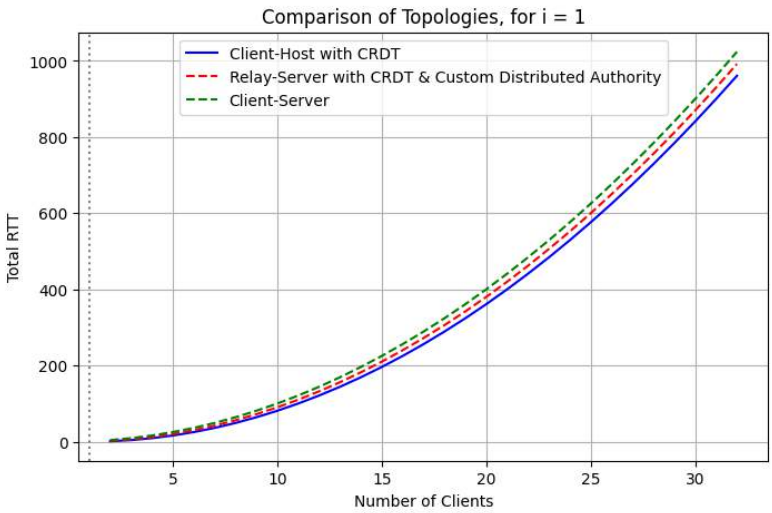

Figure 5.20: Difference between total RTT in topologies | i = 1

The results seen in Figure 5.20 are in fact only achieved when the iteration count is fixed at one. If the number of clients is fixed instead, and we vary the amount of iterations between 1 and 1000, the results of moving one operation to be treated as a CmRDT becomes more prevalent. This scenario is more typical for a general scenario that would occur in an interactive application. In interactive applications the number of operations would generally increase as the session continues, and not remain bound to one single operation. This effect can be seen in Figure 5.21. As the number of clients increase, this effective reduction yields diminishing results. The effect of increasing the amount of clients can be seen when comparing Figure 5.21 and Figure 5.22.

According to Figure 5.21 and Figure 5.22 a linear relationship is observed when increasing the amount of iterations with a set amount of clients. To understand which solution would be more effective we can consider the total RTT count more closely. When comparing the ratio between the total RTT amount of the different topologies, we fix the value $n = 8$. This ratio will tell us the proportion of messages that yield a RTT > 0. In other words, a ratio of 0.5 when dividing topology A with topology B would mean that topology A experience half as much RTT as topology B. The total amount of RTT for the `Client-Host` using CmRDTs divided by the total amount of RTT for the `Client-Server` is 0.765625. Furthermore, the total RTT for the `Client-Host` using CmRDTs divided by the total fraction of the `Relay-Server` using CmRDTs is 0.875. These results are achieved when using the value $n = 8$.

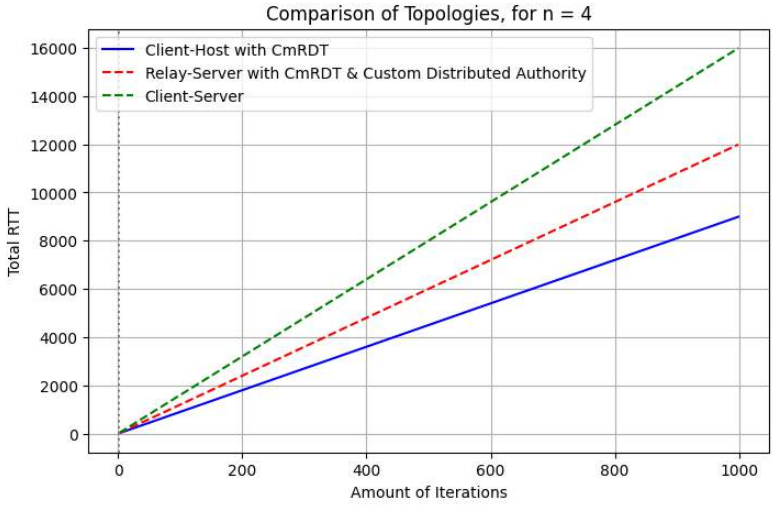

Figure 5.21: Difference in total RTT in topologies | n = 4

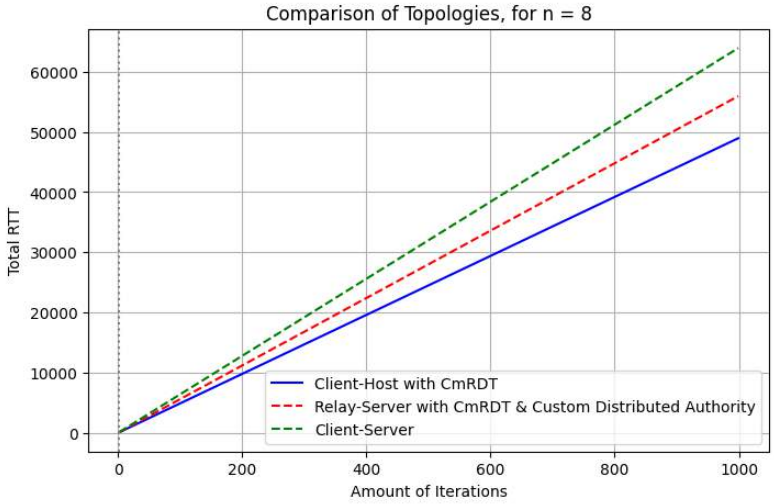

Figure 5.22: Difference in total RTT in topologies | n = 8

In summary, the observed result is that, the potential benefit, in terms of RTT reduction, diminishes over time as more clients connect to a session. However, this is not necessarily a weakness of the result, as each connecting client still experiences zero SVL. The reduction in RTT would instead be more significant when considering the total amount of traffic that has to be routed over the internet. In other words, incorporating a host as a server authority as opposed to a merely relaying node, has the added benefit of reducing the network load. This can be thought of as an optimization to an already

established local-first paradigm solution.

Chapter 6

Conclusions and Future work

This chapter links back to our original thesis topic and the research questions posed in the introduction. Section 6.1 discusses whether the results of the literature study and subsequent experiment can help answer these questions. Section 6.2 highlights what prevented us from achieving the results we were expecting. Section 6.3 explains what could be done in the future of the field of combining XR and CRDTs. Section 6.4 links this work to the goals set up by the United Nations (UN), underlying different social, economical, and environmental factors that this thesis benefit.

6.1 Conclusions

As presented in the introduction, our hypothesis was defined as: ”By inspecting interactive applications we observe that many operations can be moved outside the server and moved locally to be CRDT operations. These CRDT operations will help reduce the SVL for all session users.”. This also coincides with our first research question: ”How can the implementation of CRDTs help reduce SVL in the current XR computing frameworks used for making interactive applications?”. In our benchmark we used NGO to test one specific framework.

The outcome of this thesis shows that by introducing CRDTs as part of a local-first paradigm to Unity, we have achieved what we initially set out to accomplish. This solution consisted of combining a well-established `Client-Server` topology where the server component simply act as a `Relay-Server`, and all updates are synchronized under a local-first paradigm. We have shown that it is possible to achieve a zero SVL, or

at least close-to-zero SVL if the `Overhead Latency` is considered a meaningful delay. This is possible due to the inherent properties present in CRDTs that allow us to synchronize states with a foundational presumption of eventual consistency and strong convergence. Knowing that all operations will converge in a consistent manner, allows us to circumvent the server authority as a central component for maintaining a consistent state. As a result of introducing local-first computing, zero SVL becomes a foundational property under this paradigm. This provides significant benefits for interactive applications where latency is of big concern, most notably in the realm of XR applications where users are more sensitive to the result their own actions being delayed.

Another interesting point to consider with regards to the first research question, is the effect on server load and RTT distribution when implementing local-first computing in current XR computing frameworks. As for server load, no noticeable difference in server load could be observed through the scope of our testing. However, more iterations would have to be performed in order to corroborate any potential differences of the performance between the topologies. As for the RTT distribution, by introducing CRDTs, all clients in a session are allowed to perform local-first updates. This affect the way updates are propagated throughout a topology, and hence the RTT distribution. This is due to validation from a server authority no longer being necessary. In the benchmarks, we showed that the reduction received by introducing CRDTs directly lowered the SVL of all clients in a session.

The second research question, posed at the beginning of this thesis was: ”What limitations are there in implementing CRDTs in current XR computing frameworks?”. This also coincides with the third research question: ”How could CRDTs be integrated seamlessly into existing XR computing frameworks?”. By answering the first question, we gain insight into what needs to be done in order to integrate CmRDTs in current XR computing frameworks. One limitation that arises when implementing CmRDTs in NGO is causal order delivery. NGO, as it stands now, does not offer causal order delivery for `NetworkObjects`. As such, any solution that wishes to incorporate CmRDTs must make use of trivially commutative CmRDTs. Alternatively, if a trivially commutative object cannot be identified, the only other solution would be introduce causal order delivery via including e.g., vector clocks. As it stands now, although they were implemented in [45], we do not know if it possible to extend NGO’s functionality to include causal order

delivery in a practical manner. As mentioned earlier in Section 2.4.3.1, this might also increase memory and messaging overhead. As a result, this poses a significant limitation on the scope of implementing CmRDTs in Unity's NGO.

Additionally, another property of CmRDTs that is important to consider is idempotency. In CmRDTs, the *effect-update* function has to be idempotent, unlike in CvRDTs where the idempotency is upheld in the merge function. As mentioned in Section 2.6, RPCs in NGO makes no mention of idempotency. Therefore, developers have to ensure that this property is upheld. To solve this, a unique identifier can be attached to each operation, with a data structure that keeps track of already received RPCs and ensures they are only applied once.

Furthermore, one limitation of using RPCs as a method of synchronizing data, is that if a client connects after a given RPC is sent, they will never receive the RPC. This yields RPCs unreliable in CRDT solutions for sessions where a client can connect mid-session, where no further logic is implemented to handle this situation. To solve this, a quorum could be introduced to update new clients with the proper data. In the case of a strict P2Ps solution, all operations would have to be modified to fit CRDTs. This is due to there being no server authority present in this topology to handle state synchronization. Therefore, all operations would have to obey the properties of CRDTs to ensure consistency.

As a closing thought, common pitfalls of server-authoritative solutions are no longer present in this local-first paradigm. One of these pitfalls is the need for complements such as client-side prediction to solve the issue of latency induced by needing to wait for a server to authorize user actions. In a local-first paradigm, such as with the use of CmRDTs, the need for server authorization for user actions are completely removed. As such, there is no longer any need for complements such as client-side prediction that can incur unnecessary rollbacks. Instead, since all updates are known to converge, the user is simply free to perform its own update and share its operations to all other clients. In this way, we have not only reduced the communication load between nodes, but we have completely circumvented the server without negatively affecting the performance of the game. Additionally, we have shown that clients that operate under a local-first paradigm also experience zero SVL related to their operations. This result offers significant advantages for areas where this delay is crucial to user experience. Therefore, with these findings, by performing this benchmark we have strengthened our initial hypothesis. Despite this, one

posed research question still remain unanswered. The effect on server load of a CRDT solution could not be decisively answered.

6.2 Limitations

In the results we were hoping to observe any signs of bottleneck occurring. The benchmark was run on a cloud server with 12 CPU cores and 80 GB of RAM memory. This allowed us to run as many as 16 clients. When exceeding this, the CPU was heavily burdened handling all clients. Therefore, one limitation of the test was indeed the amount of clients that could be included. This may be a reason as to why no signs of a bottleneck could be observed. It remains that one limitation of the benchmark definitely was the amount of clients that could be run on one system. This could potentially be the resulting factor in not being able to determine the server load.

Another limitation of this thesis is the benchmarking factor. Although benchmarks were performed iteratively, in an increasing manner, the same test under the same conditions was not repeated to yield a more representative result of said conditions. Therefore, the benchmark could have been repeated many times for each $\{N, i\}$ pair to yield more accurate results in a system that was not focused solely on determining the RTT of the topology. For example, with more timestamps documenting the workflow of the server, further insights into whether a bottleneck was starting to form in any of the topologies could have been explored.

Much of the functionality of the `Distributed Authority` and `Relay-Server` topologies of NGO was restricted in access. This is due to the fact that they cost money to use. Therefore, to work around this, custom implementations that simulate these environments had to be created. As a result, we cannot say that this benchmark accurately represents NGO's `Distributed Authority` topology. To simulate this we used a `Client-Host` solution which closely resembles a stripped down version of `Distributed Authority` where the host is session owner and object owner for all objects. Since the benchmark uses one object, and changing ownership for this object for all interactions would be rather inefficient, we opted for no transfer of ownership to take place. Therefore, the host remains session owner, and object owner throughout the benchmark. As a result of the benchmark using this custom implementation, this becomes a limitation of the thesis. This is also true for the `Relay-Server`, where we instead used a `Client-Server` topology

which does not control the game state. The server instead forwards all messages to other clients. This represents an additional limitation of the thesis.

Lastly, an interesting point of discussion is the scenario in which the benchmark never terminates. This has shown to be a limitation in certain scenarios where the environment is scaled up to handle many clients. We observed during the experimental phase that some processes sleep (clients, etc.) during runtime. We hypothesize that, in this scenario, since more processes are created, the sleeping time and frequency increases as the CPU has to handle an increased number of processes.

Our benchmarking environment is designed in such a way that our data structures used to gather the timestamps expect a given number of RPCs depending on the values for $\{N, i\}$. In other words, the benchmark only terminates when all RPCs have been delivered to all replicas. In NGO, reliable delivery is only guaranteed for connected nodes, meaning if a client connects mid-session they will not be given any previous RPCs that were sent before their joining. However, consider the scenario in which a process (node) is joined from the start, and sleeps sometime during execution. In this case, NGO will drop the connection to the sleeping node after a fixed interval known in NGO as *Disconnect Timeout MS*. After this timeout has elapsed, the sleeping node will be considered disconnected. Based on Unity documentation, we know that: ”An RPC call made without an active connection won’t be automatically added to the send queue and will be dropped/ignored.” [53]. Based on the Unity documentation, in such a case, our benchmark would never terminate and it is considered a failure.

As mentioned in the first paragraph of this section, we mentioned the CPU being the factor for why we did not extend the benchmark beyond 16 clients. As the underlying aims of this thesis were to investigate SVL with the help of CRDTs, we remained satisfied with a maximum of 16 clients in our benchmark and did not investigate into a conclusive reason. That being said, the event of a client disconnecting mid-session constitutes a significant limitation in reliable delivery with RPCs in NGO. Admittedly, a scenario in which one machine hosts many nodes would most likely not be realistic as users usually run a game instance on their own machine. However, it also highlights the reality of RPCs in NGO only being reliable for active nodes. As a result, we cannot with any certainty conclude that NGO—and by extension—this implementation is tolerant to network partitions, if, by network partition we mean the scenario

in which messages cannot get across due to unavailable processes causing a loss of connectivity. This would have implications for convergence, as in SEC, eventual delivery (2.1) guarantees that *"an update delivered at some correct replica is eventually delivered to all correct replicas."* [7] However, this conditions hinges on the assumption that a network partition eventually heals and messages will eventually flow, at which updates may be delivered again. However, in this line reasoning, we have just hypothesized an instance in which processes sleep to the point where the network never fully recovers and messages are lost. As such, convergence—and by extension SEC—even in the presence of CRDTs could not be guaranteed. As a consequence, to guarantee SEC via CRDTs in NGO in general, additional assumptions on the network would be needed. Under the hypothesis of increased sleeping time and frequency associated with an increase of nodes for a given machine, and SEC no longer being guaranteed, a trade-off between scalability and consistency emerges. Although, as Shapiro mentions in [7], SEC provides a work-around for the CAP problem due to the inherent properties of CRDTs, the hypothesized scenario just described shows how consistency no longer is possible as the number of processes increased. As such, during a network partition, given that a threshold of the number of processes a machine can handle has been reached, then consistency is no longer guaranteed.

6.3 Future work

This benchmark was performed for one type of transaction, namely moving a sphere in 3D space. There are many types of transaction that occur between clients in a game environment. This benchmark could be scaled up to simulate the effect of moving operations to CRDTs in a larger environment. Furthermore, the amount of operations that could be treated as CRDTs in a current video game is an intriguing subject. The more operations that can be moved to a local-first update paradigm, the bigger the benefit of this paradigm, and a smoother the experience for the user. This is especially apparent in XR where a user's own actions being delayed often results in an uncomfortable experience.

Furthermore, a usability study could be done where users interact with a system using CRDTs opposed to one not using CRDTs. This type of study could determine what actual effect this type of solution has on user experience. Similar studies have been done within the fields of XR, although not incorporating CRDTs.

An analysis of the cost of transitioning data types—whether currently treated as CRDTs or not—to proper CRDT implementations could help a company evaluate the potential benefits of adopting CRDTs to enhance user experience. For determining this, a tool for automatization could be created for identifying possible candidates of data types that fulfills the requirement for being considered a CRDT.

CRDTs could introduce certain security risks, especially if being performed in a P2P-connected environment. A security analysis of CRDTs could be done to assess in what type of environment these data types would be viable in and what type of underlying server structures could ensure security. Furthermore, if we consider cheating in video games as another security risk, what types of security risks do CRDTs introduce in a game environment?

A benchmark could be done to determine server load when moving more and more operations to CRDTs. This could help determine the impact of having a game consisting of a certain fraction of operations being updated in a local-first manner.

Lastly, not all operations fulfill the requirements of CRDTs. In other words, it is unlikely that a game strictly would only include data types that fulfill these requirements. In such a case, it would be interesting to see what it means for a game to include both CRDT and non-CRDTs and its implications for synchronizing states under different paradigms and consistency models within an application.

6.4 Reflections on Sustainable Development

This thesis strives to reduce latency, mainly by reducing the traffic sent over the internet. This has implications for power consumption. By lowering power consumption this thesis supports goal 7 of the UN's Sustainable Development Goal (SDG), sustainable energy for everyone. In the long term, this also leads to the support of goal 13, climate action.

CRDTs allow the clients to work offline and then sync up after being offline. Several parts of the world do not have a stable internet connection, and therefore, cooperating on projects in various interactive applications do not work as smoothly as for others. However, due to the convergence properties

of CRDTs, users are able to work offline and later merge their states without conflict. Therefore, this thesis contributes to goal 9 of the SDG by providing access to these types of applications in parts of the world where unstable internet connection remains a reality. As a result, this work also contributes to goal 10 of the SDG by reducing inequalities between various parts of the world.

References

[1] K. Brunnström et al., "Latency impact on quality of experience in a virtual reality simulator for remote control of machines," *Signal Processing: Image Communication*, vol. 89, p. 116005, 2020. doi: https://doi.org/10.1016/j.image.2020.116005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0923596520301648 [Page 1.]

[2] A. Dantas and C. Baquero, "Crdt-based game state synchronization in peer-to-peer vr," in *Proceedings of the 12th Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '25. New York, NY, USA: Association for Computing Machinery, 2025. doi: 10.1145/3721473.3722144. ISBN 9798400715587 p. 45–55. [Online]. Available: https://doi.org/10.1145/3721473.3722144 [Pages 1, 3, and 29.]

[3] M. Alsakati, "Wi-fi 7: Multi-link operation in xr industrial scenarios," Master's thesis, KTH, Royal Institute of Technology, EECS, Stockholm, Sweden, August 2022. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-323195 [Page 1.]

[4] S. M. Rumble et al., "It's time for low latency," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. USA: USENIX Association, 2011, p. 11. [Page 1.]

[5] M. Vasarainen et al., "A systematic literature review on extended reality: Virtual, augmented and mixed reality in working life," *International Journal of Virtual Reality*, vol. 21, no. 2, pp. 1–28, 2021. doi: 10.20870/IJVR.2021.21.2.4620. [Online]. Available: https://doi.org/10.20870/IJVR.2021.21.2.4620 [Page 1.]

[6] Unity Technologies, “High level networking concepts,” https://docs.unity3d.com/400/Documentation/Components/net-HighLevelOverview.html, 2014, accessed: 2025-05-22. [Pages 2 and 9.]

[7] M. Shapiro et al., “Conflict-free replicated data types,” in *Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. ISBN 978-3-642-24550-3 pp. 386–400. [Pages 2, 5, 15, 16, 17, 20, 21, 22, 23, 24, 25, and 78.]

[8] M. Barbosa et al., “Secure conflict-free replicated data types,” in *Proceedings of the 22nd International Conference on Distributed Computing and Networking*, ser. ICDCN ’21. New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3427796.3427831. ISBN 9781450389334 p. 6–15. [Online]. Available: https://doi.org/10.1145/3427796.3427831 [Page 5.]

[9] C. Cortés et al., “Impact of self-view latency on quality of experience: Analysis of natural interaction in xr environments,” in *2022 IEEE International Conference on Image Processing (ICIP)*, 2022. doi: 10.1109/ICIP46576.2022.9897983 pp. 3131–3135. [Pages 7 and 29.]

[10] A. Reeve, “Unity documentation: Network topologies,” Oct 2024, accessed: 2025-02-05. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/terms-concepts/network-topologies/ [Pages 9, 10, and 11.]

[11] ——, “Unity documentation: Latency and packet loss,” jan 2024, accessed: 2025-02-05. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/1.6.0/learn/lagandpacketloss/ [Page 9.]

[12] S. Sulyman, “Client-server model,” *IOSR Journal of Computer Engineering*, vol. 16, pp. 57–71, 01 2014. doi: 10.9790/0661-16195771 [Page 9.]

[13] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 7th ed. Boston, MA: Pearson, 2017. ISBN 9780133594140 [Page 9.]

[14] V. Cooper, “Unity documentation: Client-server topologies,” Mar 2025, accessed: 2025-06-03. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/terms-concepts/client-server/ [Page 10.]

[15] N. Stephens, “Unity documentation: Understanding ownership and authority,” Jun 2025, accessed: 2025-06-05. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/basics/ownership/ [Page 10.]

[16] Photon, “Network topologies,” accessed: 2025-06-05. [Online]. Available: https://doc.photonengine.com/fusion/current/manual/network-topologies [Pages 10 and 11.]

[17] F. Ehiagwina et al., “A state-of-the-art survey of peer-to-peer networks: Research directions, applications and challenges,” vol. 1, pp. 19–38, feb 2022. doi: 10.55708/js0101003 [Page 11.]

[18] K. Leibnitz et al., “Peer-to-peer vs. client/server: Reliability and efficiency of a content distribution service,” in *Managing Traffic Performance in Converged Networks*, L. Mason, T. Drwiega, and J. Yan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1161–1172. [Page 11.]

[19] L. Lu et al., “Performance evaluation and simulation of peer-to-peer protocols for massively multiplayer online games,” *Multimedia Tools Appl.*, vol. 74, no. 8, p. 2763–2780, Apr. 2015. doi: 10.1007/s11042-013-1662-y. [Online]. Available: https://doi.org/10.1007/s11042-013-1662-y [Page 11.]

[20] Unity, “Use relay with netcode for gameobjects (ngo),” accessed: 2025-02-20. [Online]. Available: https://docs.unity.com/ugs/manual/relay/manual/relay-and-ngo [Page 11.]

[21] Mirror, “User manual - general,” march 2024, accessed: 2025-02-20. [Online]. Available: https://mirror-networking.gitbook.io/docs/manual/general [Page 12.]

[22] Steam, “Isteamnetworkingsockets interface,” accessed: 2025-02-20. [Online]. Available: https://partner.steamgames.com/doc/api/ISteamnetworkingSockets [Page 12.]

[23] Fish-Networking, “Fish-networking introduction,” feb 2025, accessed: 2025-02-20. [Online]. Available: https://fish-networking.gitbook.io/docs [Page 12.]

[24] M. van Steen and A. Tanenbaum, "A brief introduction to distributed systems," *Computing*, vol. 98, 10 2016. doi: 10.1007/s00607-016-0508-7 [Page 12.]

[25] M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 3rd ed. CreateSpace Independent Publishing Platform, 2017. ISBN 978-1543057386. [Online]. Available: https://www.distributed-systems.net/index.php/books/ds3/ [Pages 12 and 13.]

[26] S. Solat, "Sharding distributed databases: A critical review," 04 2024. doi: 10.48550/arXiv.2404.04384 [Pages 12, 13, 15, and 16.]

[27] P. Carbone, "Distibuted acid (the 2pc protocol)," 2025, slides available on the canvas page of the course. [Pages 13, 15, and 16.]

[28] R. Alves Campêlo, M. Casanova, D. Guedes, and A. Laender, "A brief survey on replica consistency in cloud environments," *Journal of Internet Services and Applications*, vol. 11, p. 1, 02 2020. doi: 10.1186/s13174-020-0122-y [Page 13.]

[29] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Comput. Surv.*, vol. 49, no. 1, Jun. 2016. doi: 10.1145/2926965. [Online]. Available: https://doi.org/10.1145/2926965 [Page 14.]

[30] C. Barthels, I. Müller, K. Taranov, G. Alonso, and T. Hoefler, "Strong consistency is not hard to get: two-phase locking and two-phase commit on thousands of cores," *Proc. VLDB Endow.*, vol. 12, no. 13, p. 2325–2338, Sep. 2019. doi: 10.14778/3358701.3358702. [Online]. Available: https://doi.org/10.14778/3358701.3358702 [Page 14.]

[31] V. B. F. Gomes et al., "Verifying strong eventual consistency in distributed systems," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. doi: 10.1145/3133933. [Online]. Available: https://doi.org/10.1145/3133933 [Pages 14 and 15.]

[32] P. Carbone, "Regular and atomic registers," 2024, slides available on the canvas page of the course. [Page 14.]

[33] D. J. Rosenkrantz et al., *Consistency and serializability in concurrent database systems*. Dordrecht: Springer Netherlands, 2009, pp. 99–132. ISBN 978-1-4020-9688-4. [Online]. Available: https://doi.org/10.1007/978-1-4020-9688-4_5 [Page 14.]

[34] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, p. 463–492, Jul. 1990. doi: 10.1145/78969.78972. [Online]. Available: https://doi.org/10.1145/78969.78972 [Page 14.]

[35] C. Barthels et al., "Strong consistency is not hard to get: two-phase locking and two-phase commit on thousands of cores," *Proc. VLDB Endow.*, vol. 12, no. 13, p. 2325–2338, Sep. 2019. doi: 10.14778/3358701.3358702. [Online]. Available: https://doi.org/10.14778/3358701.3358702 [Page 15.]

[36] S. Amelia and S. Ríos, "Sharding strategies and their impact on distributed database performance," *ResearchGate*, 05 2025. [Online]. Available: https://www.researchgate.net/publication/391773103_Sharding_Strategies_and_Their_Impact_on_Distributed_Database_Performance [Page 15.]

[37] Y. J. Al-Houmaily, "Atomic commit protocols, their integration, and their optimisations in distributed database systems," *International Journal of Intelligent Information and Database Systems*, vol. 4, no. 4, pp. 373–412, 2010. doi: 10.1504/IJIIDS.2010.035582. [Online]. Available: https://www.inderscience.com/info/inarticle.php?artid=35582 [Pages 15 and 16.]

[38] H. Howard and R. Mortier, "Paxos vs raft: have we reached consensus on distributed consensus?" in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, Apr. 2020. doi: 10.1145/3380787.3393681 p. 1–9. [Online]. Available: http://dx.doi.org/10.1145/3380787.3393681 [Page 16.]

[39] M. Shapiro et al., "A comprehensive study of Convergent and Commutative Replicated Data Types," Inria – Centre Paris-Rocquencourt ; INRIA, Research Report RR-7506, Jan 2011. [Pages 17, 23, 24, and 25.]

[40] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, 1985. doi: https://doi.org/10.1016/0020-0190(85)90056-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0020019085900560 [Pages 18, 19, and 20.]

[41] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond: How can applications be built on eventually

consistent infrastructure given no guarantee of safety?" *Queue*, vol. 11, no. 3, p. 20–32, Mar. 2013. doi: 10.1145/2460276.2462076. [Online]. Available: https://doi.org/10.1145/2460276.2462076 [Pages 18 and 20.]

[42] W. Vogels, "Eventually consistent: Building reliable distributed systems at a worldwide scale demands trade-offs?between consistency and availability." *Queue*, vol. 6, no. 6, p. 14–19, Oct. 2008. doi: 10.1145/1466443.1466448. [Online]. Available: https://doi.org/10.1145/1466443.1466448 [Page 20.]

[43] E. Brewer, "Cap twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012. doi: 10.1109/MC.2012.37 [Page 21.]

[44] P. Carbone, "Weaker consistency models crdts," 2024, slides available on the canvas page of the course. [Pages 22 and 24.]

[45] C. Baquero, et al., "Making Operation-Based CRDTs Operation-Based," in *Lecture Notes in Computer Science*, ser. Distributed Applications and Interoperable Systems, D. Hutchison, T. Kanade, B. Steffen, D. Terzopoulos, D. Tygar, G. Weikum, K. Magoutis, P. Pietzuch, J. Kittler, J. M. Kleinberg, A. Kobsa, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, and C. P. Rangan, Eds., vol. LNCS-8460. Berlin, Germany: Springer, Jun. 2014. doi: 10.1007/978-3-662-43352-2_11 pp. 126–140. [Online]. Available: https://inria.hal.science/hal-01287738 [Pages 25, 27, 29, and 74.]

[46] A. Reeve, "Unity documentation: Synchronizing states and events," jul 2024, accessed: 2025-02-18. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/ways-synchronize/ [Page 25.]

[47] X. Défago, P. Urban, and A. Schiper, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, 12 2004. doi: 10.1145/1041680.1041682 [Page 26.]

[48] A. Bouajjani, C. Enea, K. Ji, and S. Qadeer, *On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, 07 2018, pp. 372–391. ISBN 978-3-319-96141-5 [Page 26.]

[49] A. Reeve, "Unity documentation: Unity relay," Oct 2024, accessed: 2025-02-05. [Online]. Available: https://docs.unity.com/ugs/manual/relay/manual/introduction [Page 26.]

[50] ——, "Unity documentation: Synchronizing states and events," may 2024, accessed: 2025-02-18. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/messaging-system/ [Page 26.]

[51] A. D. Birrell and B. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, p. 39–59, Feb. 1984. doi: 10.1145/2080.357392. [Online]. Available: https://doi.org/10.1145/2080.357392 [Page 26.]

[52] A. Reeve, "Unity documentation: Rpc," may 2024, accessed: 2025-02-18. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/message-system/rpc/#rpc-targets [Page 26.]

[53] ——, "Reliability," jul 2024, accessed: 2025-02-18. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/message-system/reliability/ [Pages 27 and 77.]

[54] ——, "Networkvariables," aug 2024, accessed: 2025-02-18. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/basics/networkvariable/ [Page 28.]

[55] ——, "Unity documentation: Networktime and ticks," may 2025, accessed: 2025-06-11. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/networktime-ticks/ [Page 28.]

[56] ——, "Custom messages," feb 2024, accessed: 2025-02-18. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/message-system/custom-messages/ [Page 28.]

[57] A. Tranquillini, "Handling of mobile applications state using conflict-free replicated data types," Master's thesis, KTH Royal Institute of Technology, 2022. [Page 29.]

[58] E. Yanakieva et al., "A study of semantics for crdt-based collaborative spreadsheets," in *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '23. New

York, NY, USA: Association for Computing Machinery, 2023. doi: 10.1145/3578358.3591324. ISBN 9798400700866 p. 37–43. [Online]. Available: https://doi-org.focus.lib.kth.se/10.1145/3578358.3591324 [Page 29.]

[59] B. Ghiffar et al., "Svgdoc: A crdt for collaborative vector graphics editing," in *2024 International Symposium on Parallel Computing and Distributed Systems (PCDS)*, 09 2024. doi: 10.1109/PCDS61776.2024.10743664 pp. 1–5. [Page 29.]

[60] A. Ehlert, "Improving input prediction in online fighting games," p. 62, 2021. [Page 29.]

[61] D. Navalho et al., "A study of crdts that do computations," in *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '15. New York, NY, USA: Association for Computing Machinery, 2015. doi: 10.1145/2745947.2745948. ISBN 9781450335379. [Online]. Available: https://doi.org/10.1145/2745947.2745948 [Page 29.]

[62] M. Ptaszek, "Chat service architecture: Persistence," Jan 2016, accessed: 2025-06-11. [Online]. Available: https://technology.riotgames.com/news/chat-service-architecture-persistence [Page 29.]

[63] H. Scalability, "How league of legends scaled chat to 70 million players - it takes lots of minions." *High Scalability*, October 2014, accessed: 2025-02-20. [Online]. Available: https://highscalability.com/how-league-of-legends-scaled-chat-to-70-million-players-it-t/ [Page 29.]

[64] T. Grüne-Yanoff, *Justified Method Choice - Scientific Methodology for Scientists and Engineers*, fourth edition, preliminary version ed., 2022, revision date: 2022-10-28. [Pages 54, 56, and 57.]

Appendix A

Scatter Plots 500 and 1000 Iterations

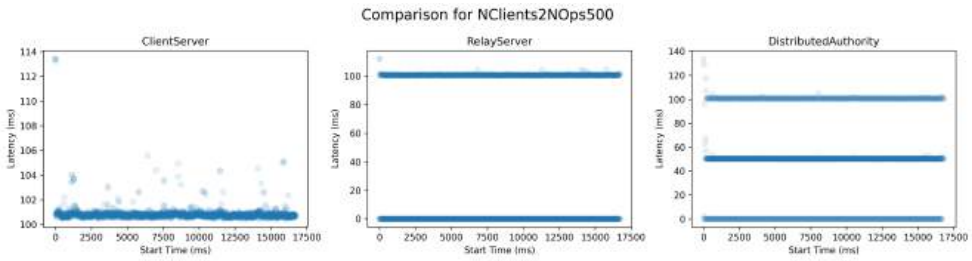

Figure A.1: | Scatter Plot | 2 Clients 500 Operations

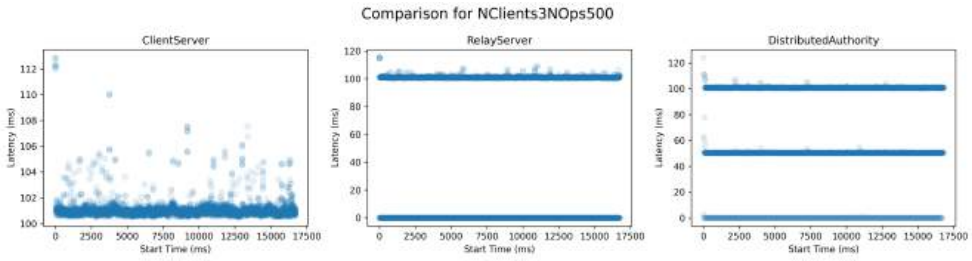

Figure A.2: | Scatter Plot | 3 Clients 500 Operations

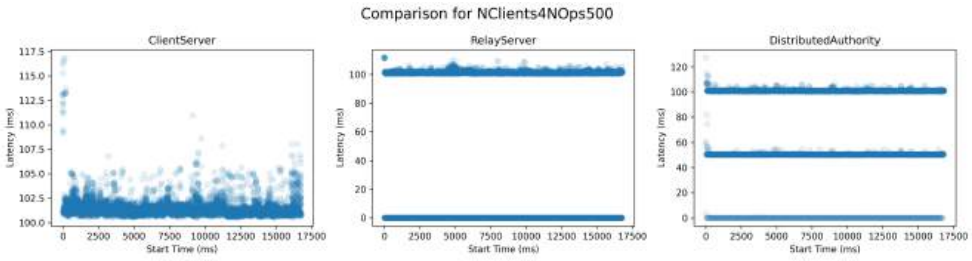

Figure A.3: | Scatter Plot | 4 Clients 500 Operations

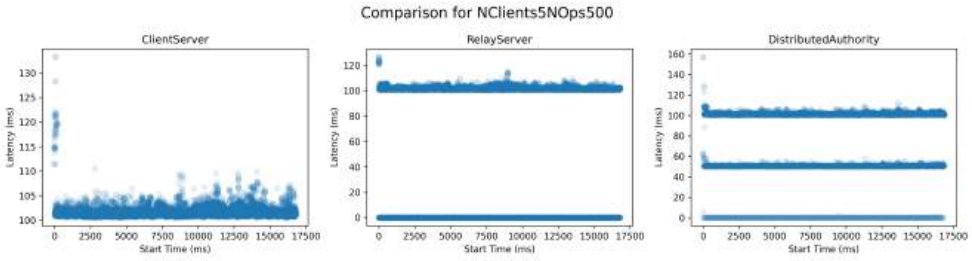

Figure A.4: | Scatter Plot | 5 Clients 500 Operations

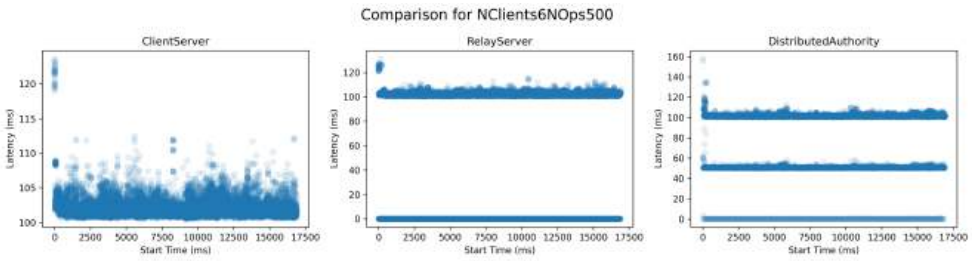

Figure A.5: | Scatter Plot | 6 Clients 500 Operations

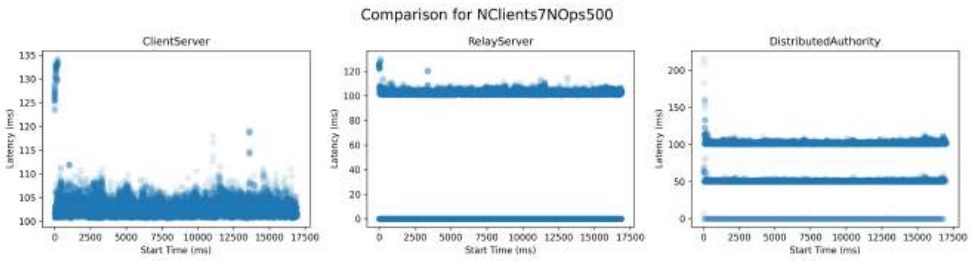

Figure A.6: | Scatter Plot | 7 Clients 500 Operations

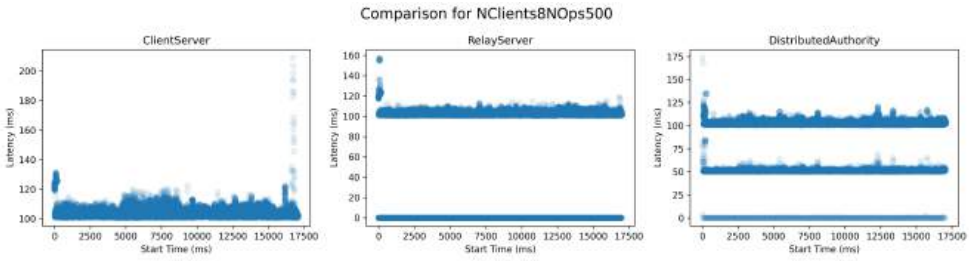

Figure A.7: | Scatter Plot | 8 Clients 500 Operations

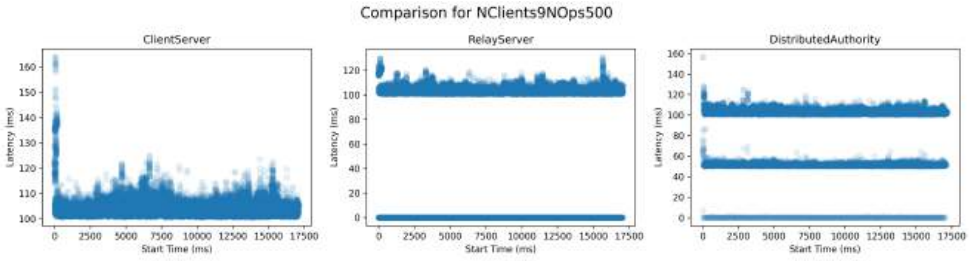

Figure A.8: | Scatter Plot | 9 Clients 500 Operations

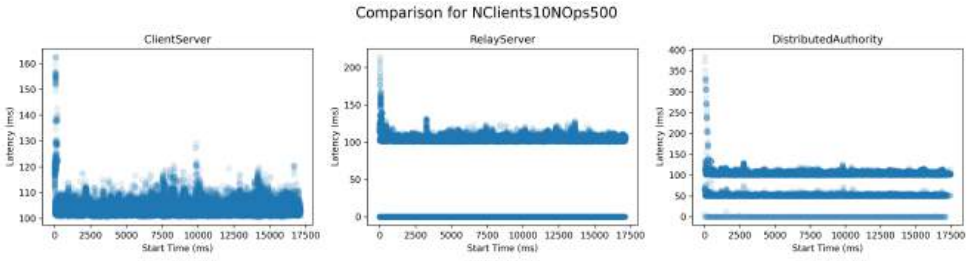

Figure A.9: | Scatter Plot | 10 Clients 500 Operations

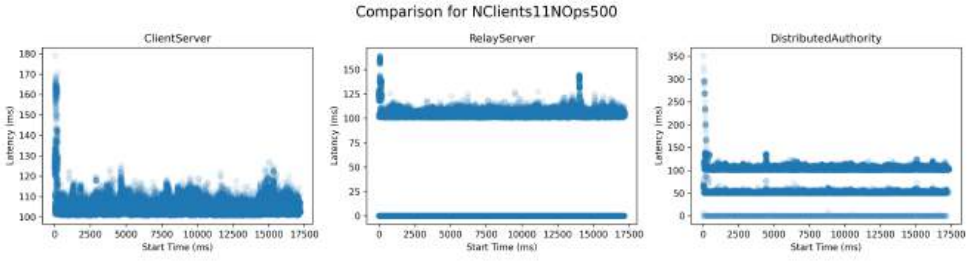

Figure A.10: | Scatter Plot | 11 Clients 500 Operations

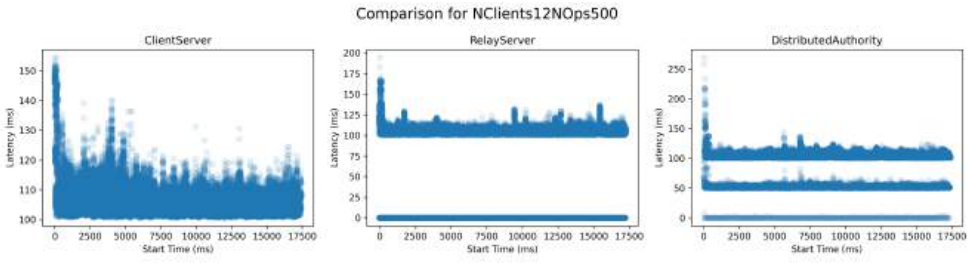

Figure A.11: | Scatter Plot | 12 Clients 500 Operations

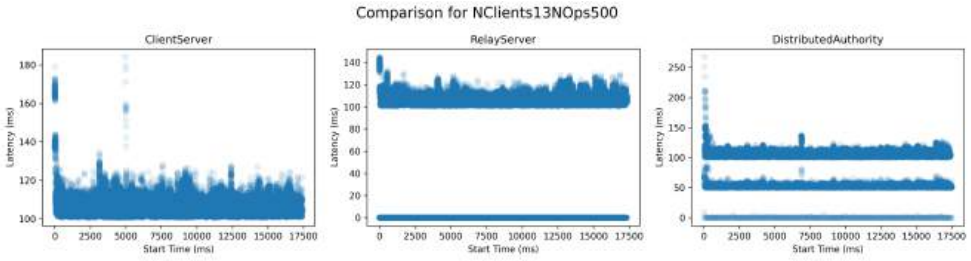

Figure A.12: | Scatter Plot | 13 Clients 500 Operations

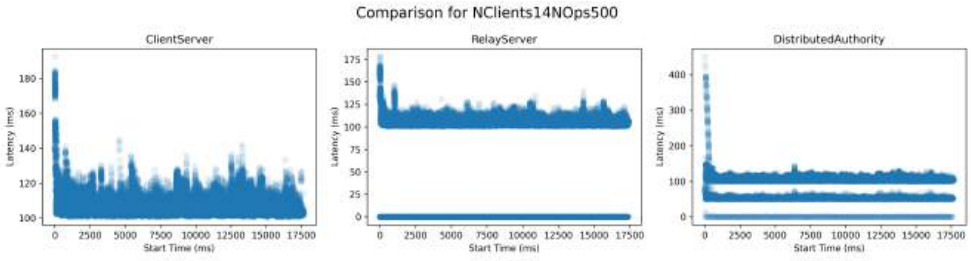

Figure A.13: | Scatter Plot | 14 Clients 500 Operations

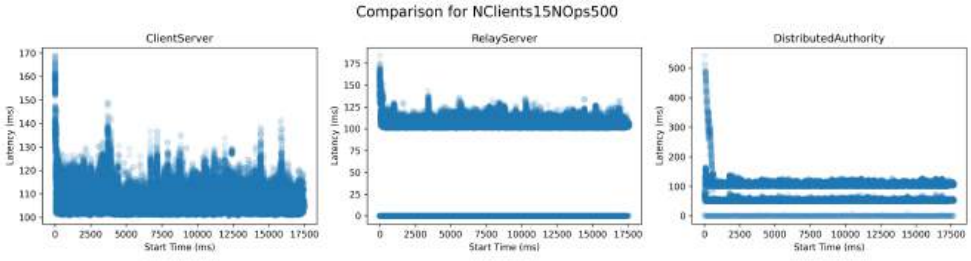

Figure A.14: | Scatter Plot | 15 Clients 500 Operations

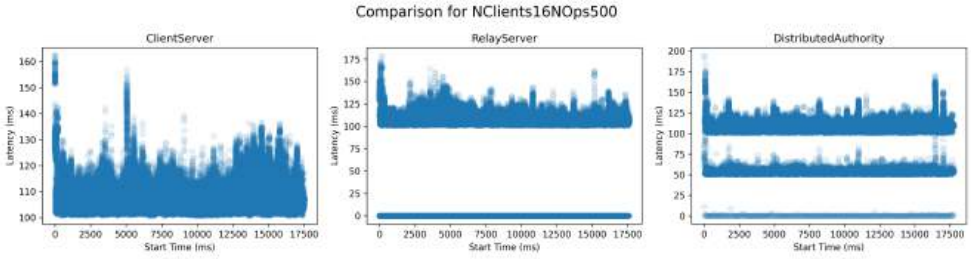

Figure A.15: | Scatter Plot | 16 Clients 500 Operations

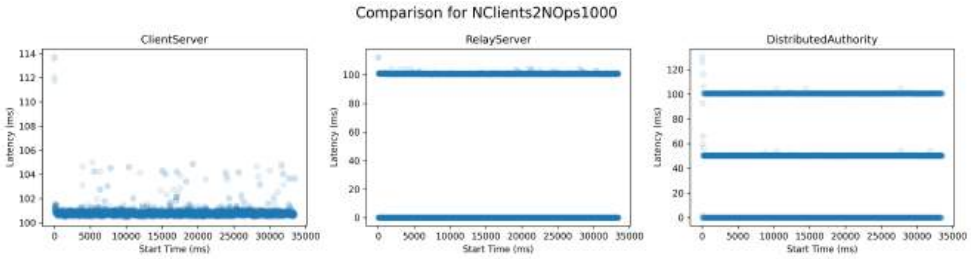

Figure A.16: | Scatter Plot | 2 Clients 1000 Operations

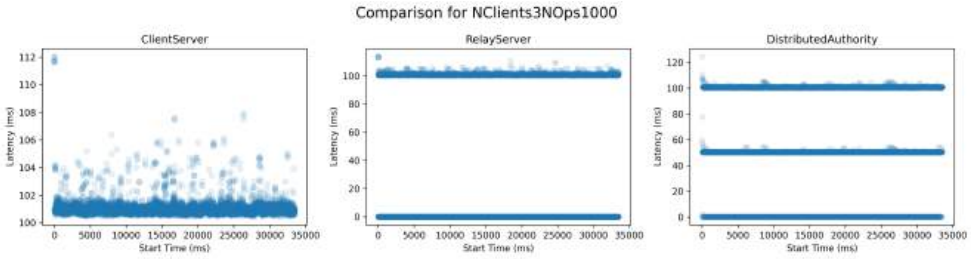

Figure A.17: | Scatter Plot | 3 Clients 1000 Operations

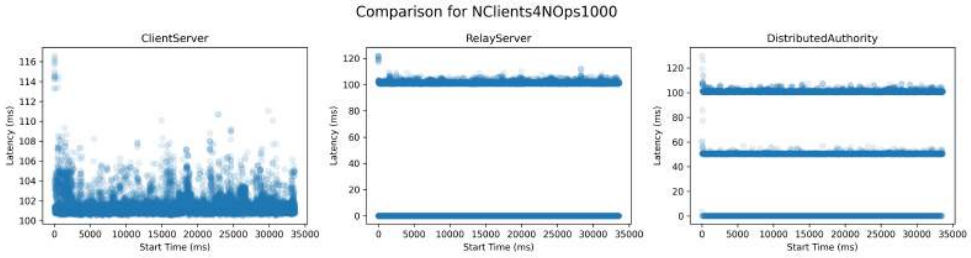

Figure A.18: | Scatter Plot | 4 Clients 1000 Operations

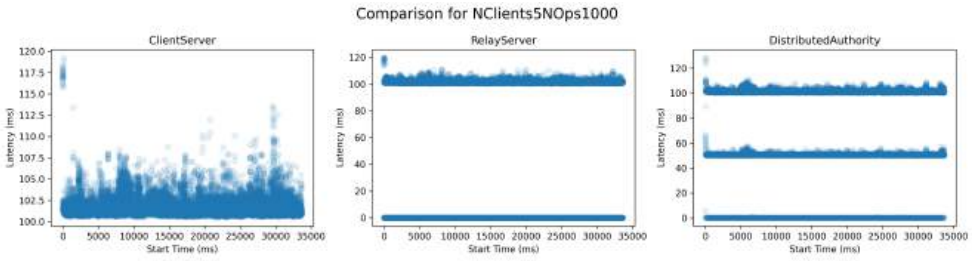

Figure A.19: | Scatter Plot | 5 Clients 1000 Operations

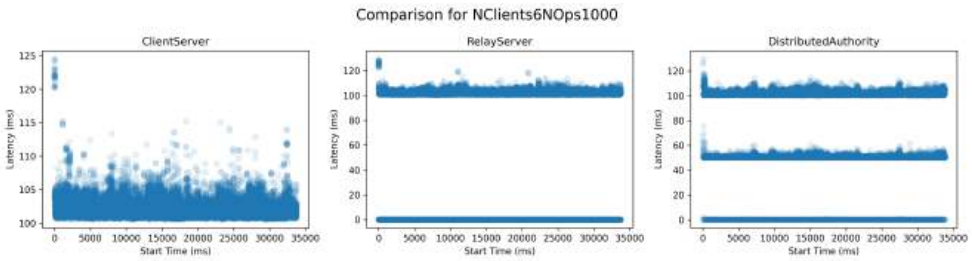

Figure A.20: | Scatter Plot | 6 Clients 1000 Operations

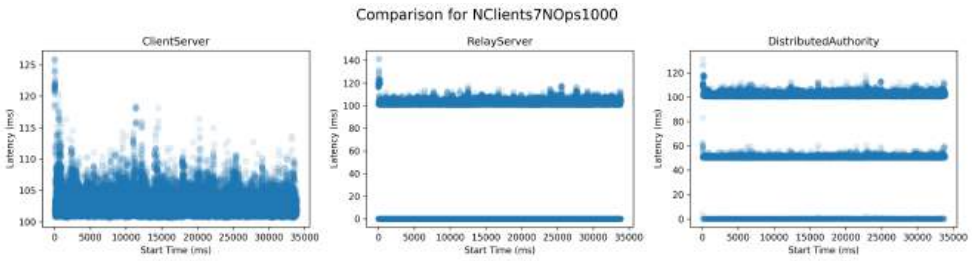

Figure A.21: | Scatter Plot | 7 Clients 1000 Operations

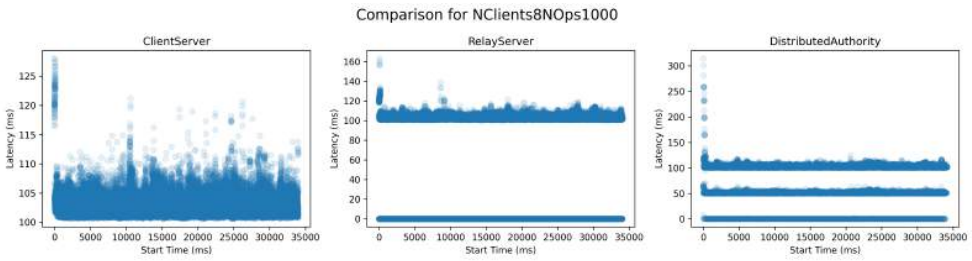

Figure A.22: | Scatter Plot | 8 Clients 1000 Operations

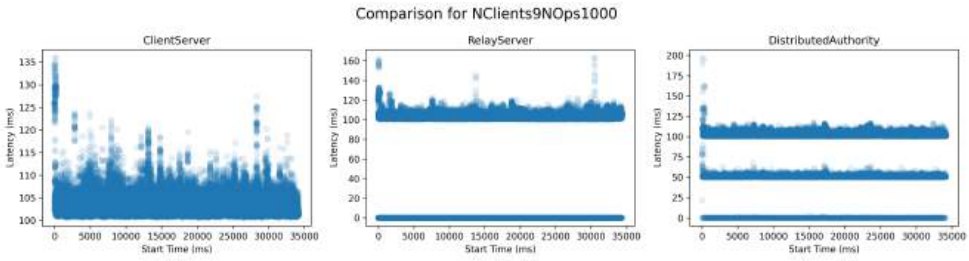

Figure A.23: | Scatter Plot | 9 Clients 1000 Operations

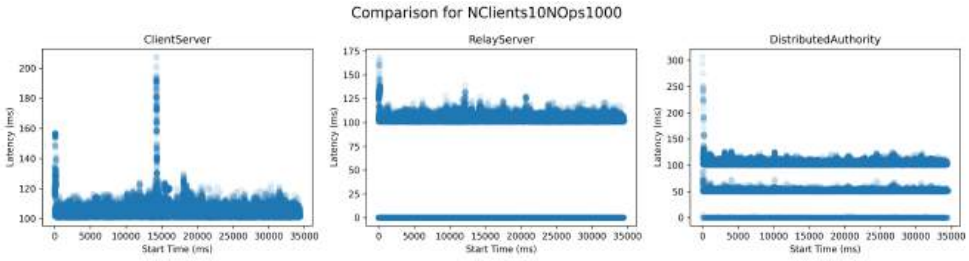

Figure A.24: | Scatter Plot | 10 Clients 1000 Operations

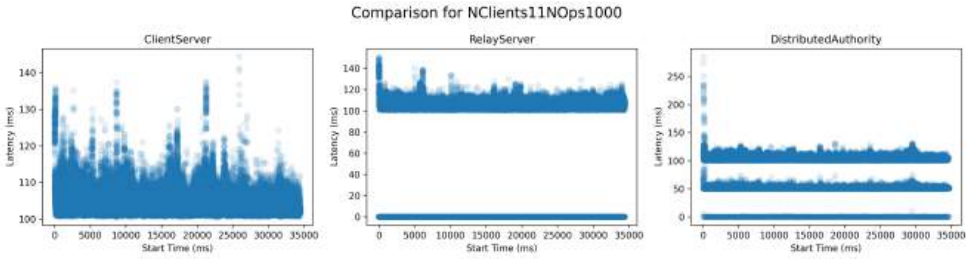

Figure A.25: | Scatter Plot | 11 Clients 1000 Operations

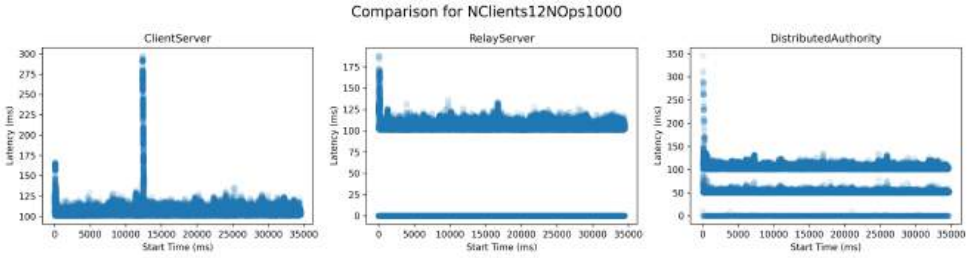

Figure A.26: | Scatter Plot | 12 Clients 1000 Operations

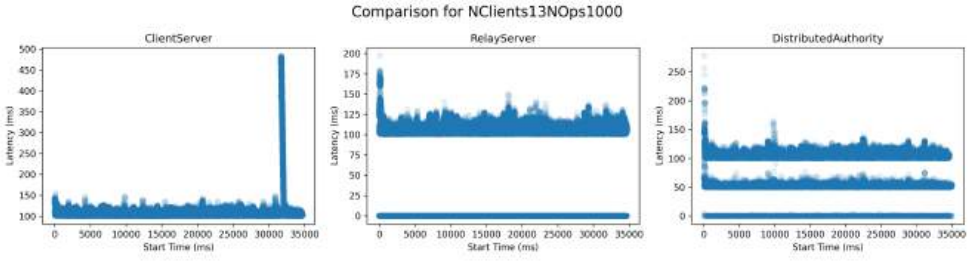

Figure A.27: | Scatter Plot | 13 Clients 1000 Operations

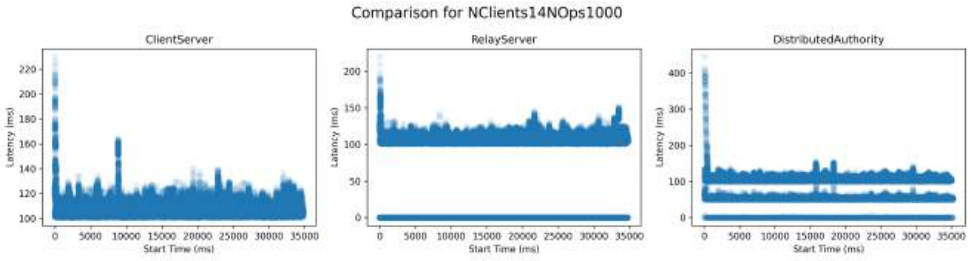

Figure A.28: | Scatter Plot | 14 Clients 1000 Operations

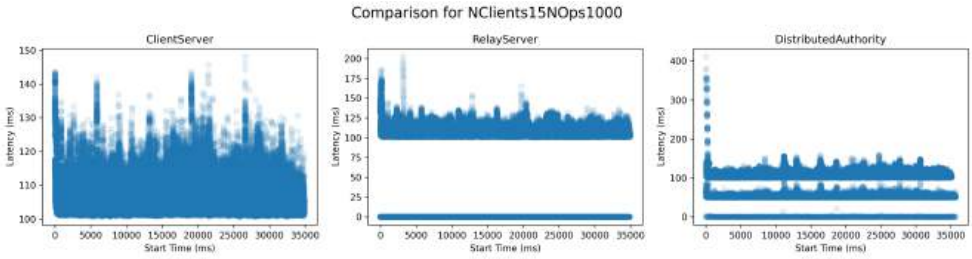

Figure A.29: | Scatter Plot | 15 Clients 1000 Operations

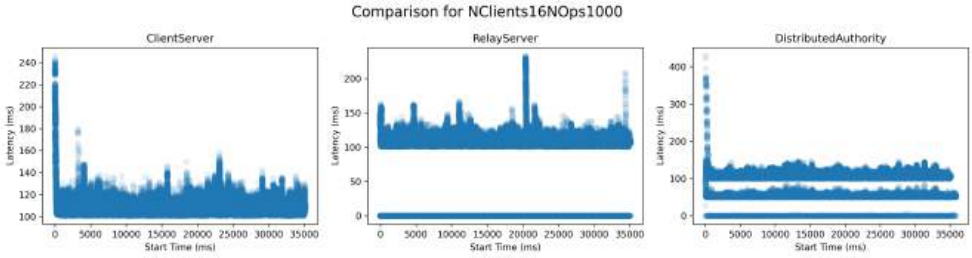

Figure A.30: | Scatter Plot | 16 Clients 1000 Operations

Appendix B

Scatter Plots 1500 and 2000 Iterations

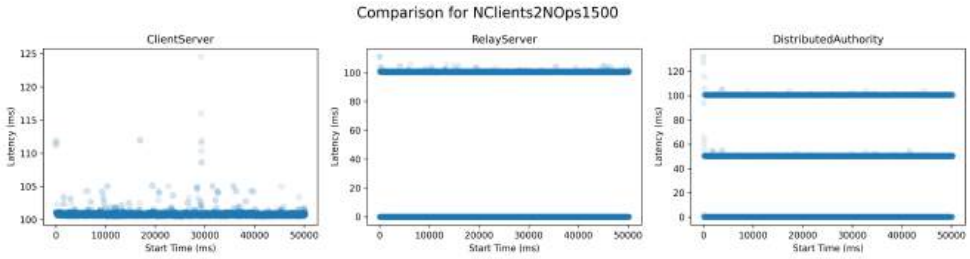

Figure B.1: | Scatter Plot | 2 Clients 1500 Operations

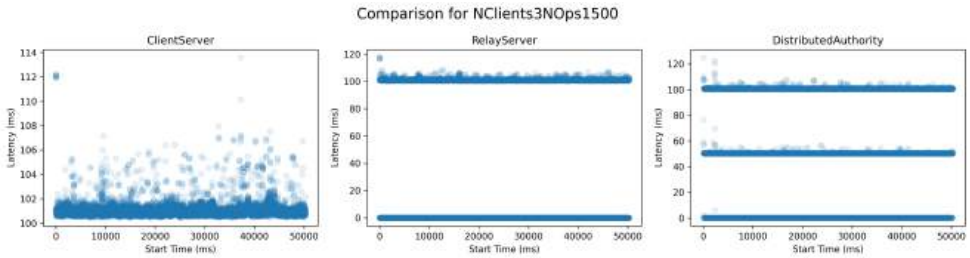

Figure B.2: | Scatter Plot | 3 Clients 1500 Operations

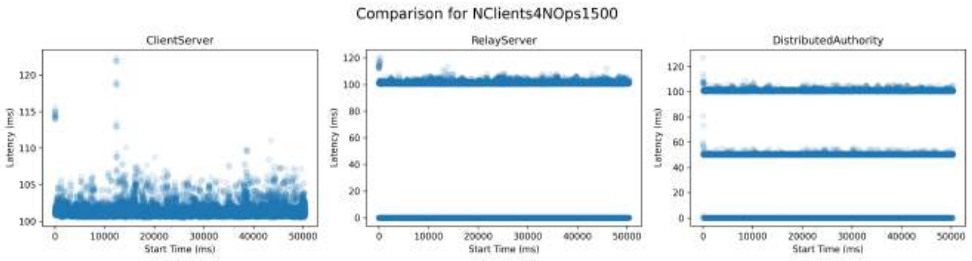

Figure B.3: | Scatter Plot | 4 Clients 1500 Operations

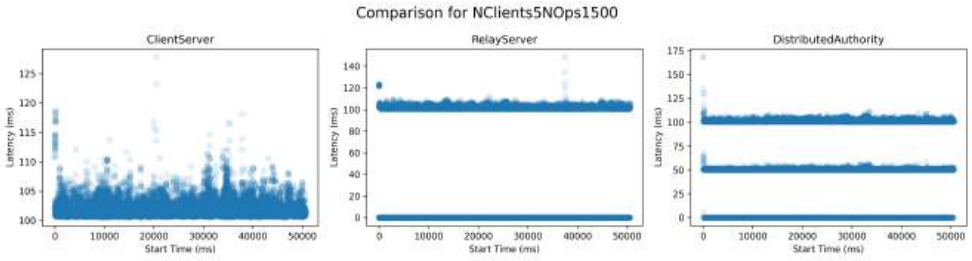

Figure B.4: | Scatter Plot | 5 Clients 1500 Operations

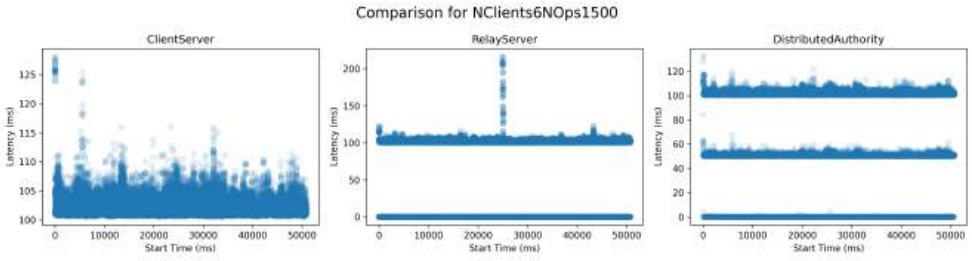

Figure B.5: | Scatter Plot | 6 Clients 1500 Operations

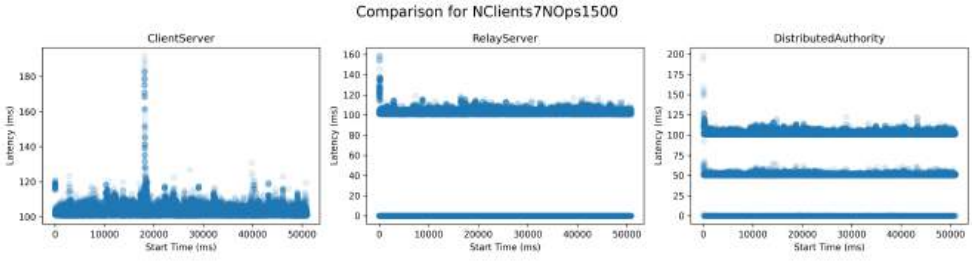

Figure B.6: | Scatter Plot | 7 Clients 1500 Operations

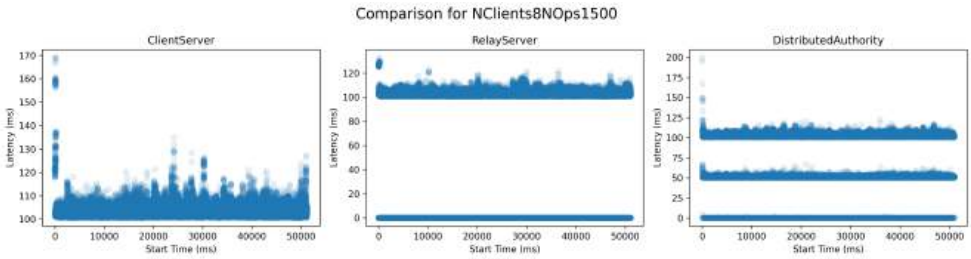

Figure B.7: | Scatter Plot | 8 Clients 1500 Operations

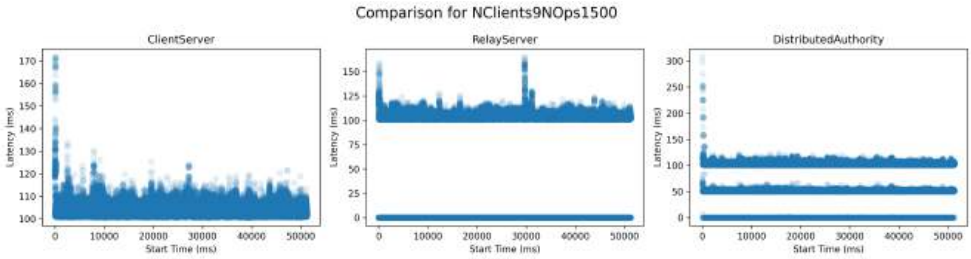

Figure B.8: | Scatter Plot | 9 Clients 1500 Operations

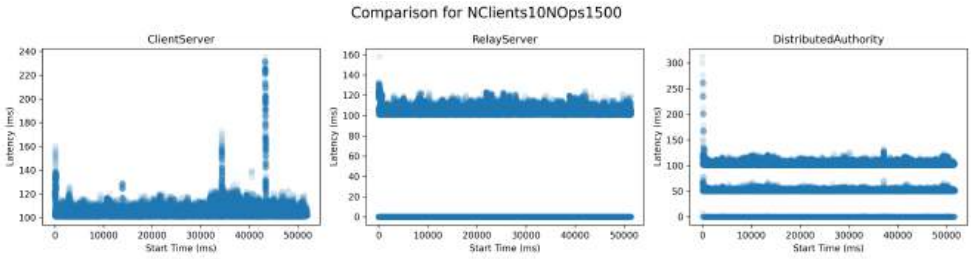

Figure B.9: | Scatter Plot | 10 Clients 1500 Operations

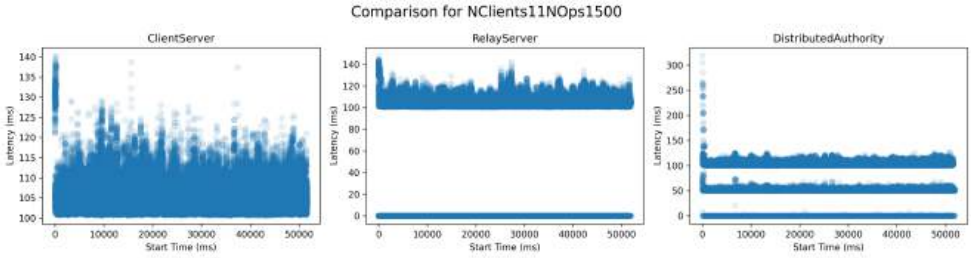

Figure B.10: | Scatter Plot | 11 Clients 1500 Operations

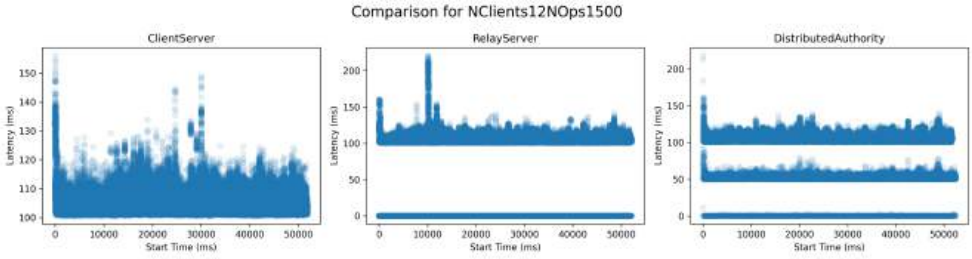

Figure B.11: | Scatter Plot | 12 Clients 1500 Operations

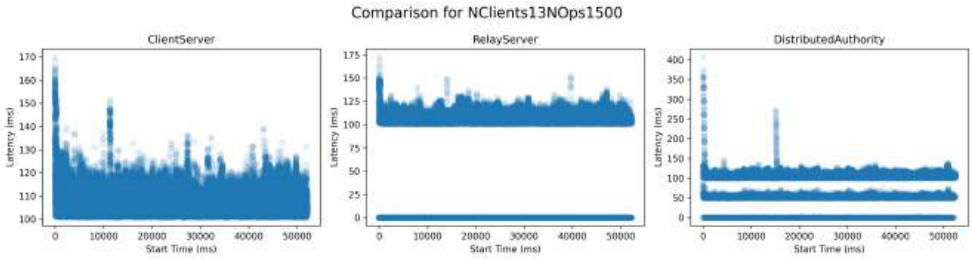

Figure B.12: | Scatter Plot | 13 Clients 1500 Operations

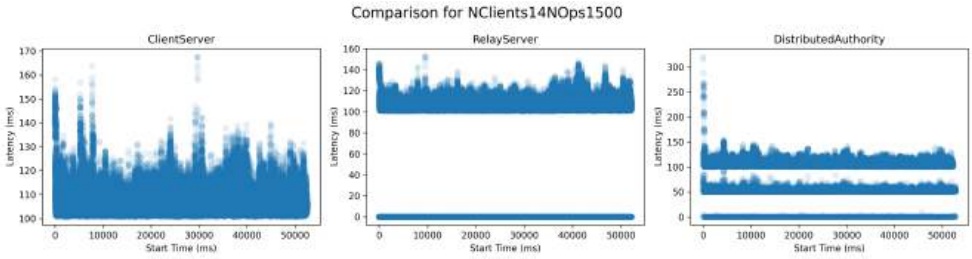

Figure B.13: | Scatter Plot | 14 Clients 1500 Operations

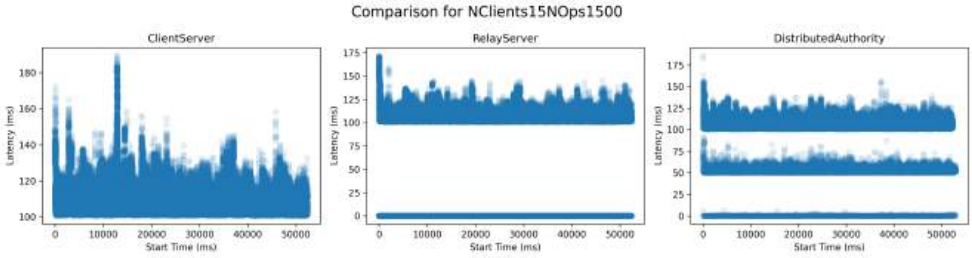

Figure B.14: | Scatter Plot | 15 Clients 1500 Operations

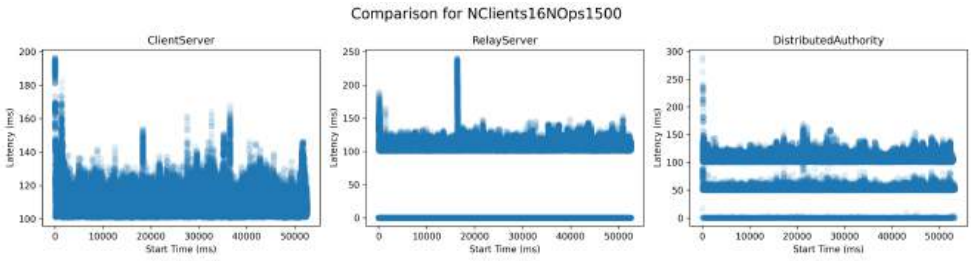

Figure B.15: | Scatter Plot | 16 Clients 1500 Operations

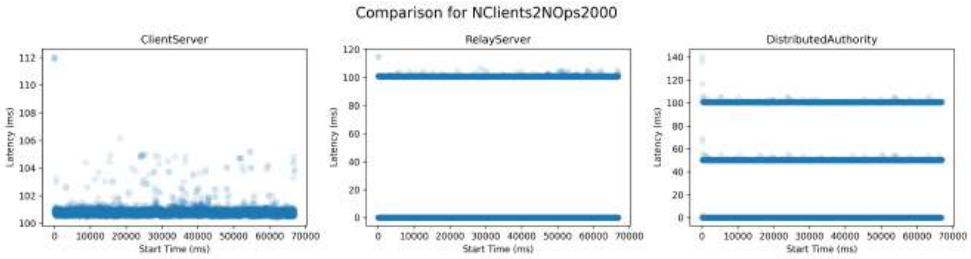

Figure B.16: | Scatter Plot | 2 Clients 2000 Operations

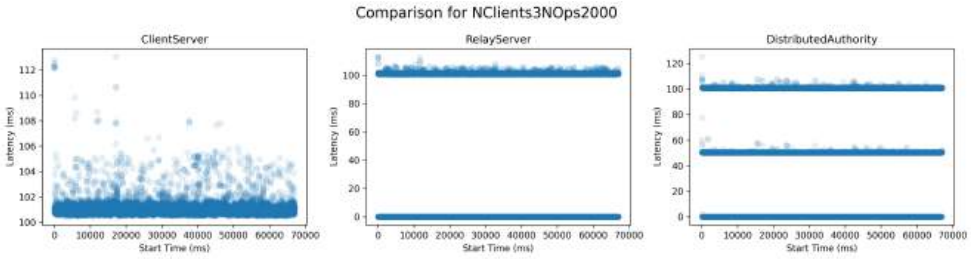

Figure B.17: | Scatter Plot | 3 Clients 2000 Operations

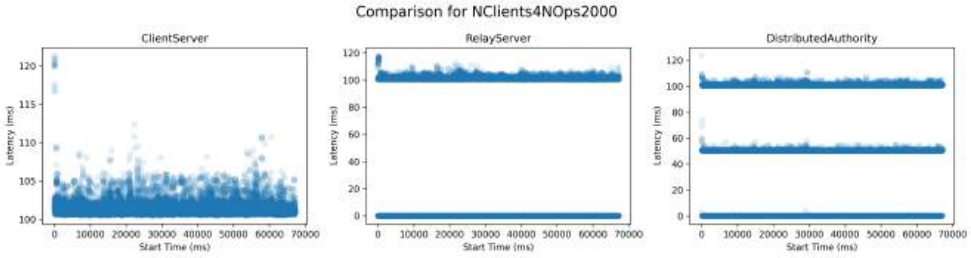

Figure B.18: | Scatter Plot | 4 Clients 2000 Operations

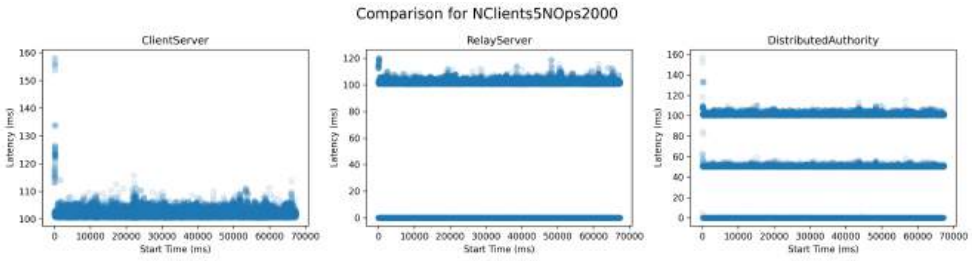

Figure B.19: | Scatter Plot | 5 Clients 2000 Operations

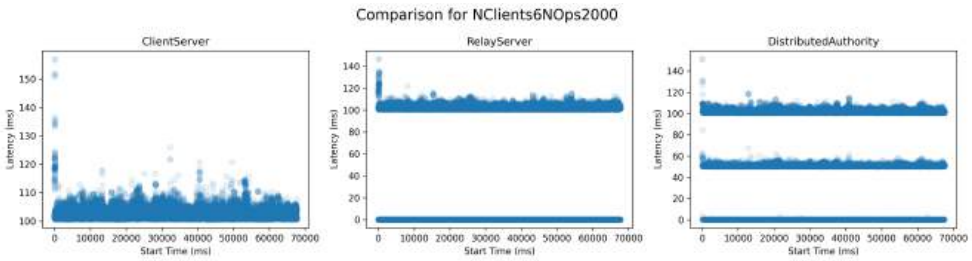

Figure B.20: | Scatter Plot | 6 Clients 2000 Operations

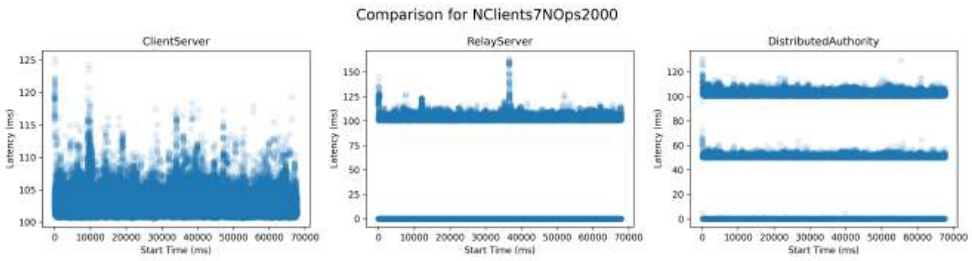

Figure B.21: | Scatter Plot | 7 Clients 2000 Operations

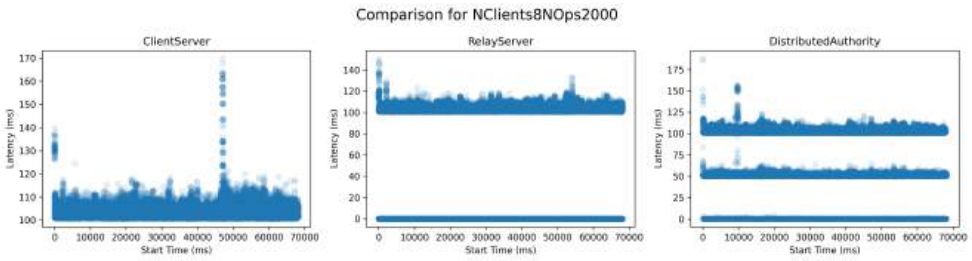

Figure B.22: | Scatter Plot | 8 Clients 2000 Operations

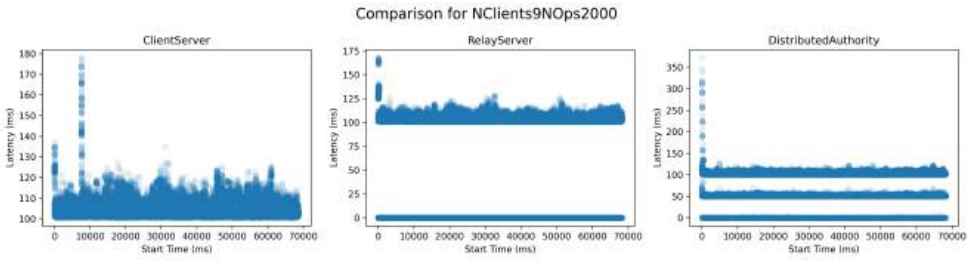

Figure B.23: | Scatter Plot | 9 Clients 2000 Operations

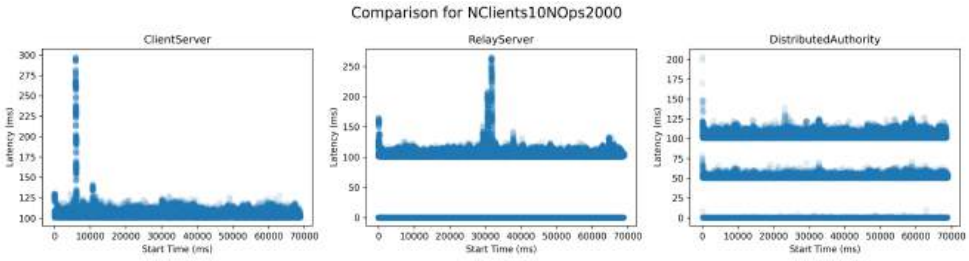

Figure B.24: | Scatter Plot | 10 Clients 2000 Operations

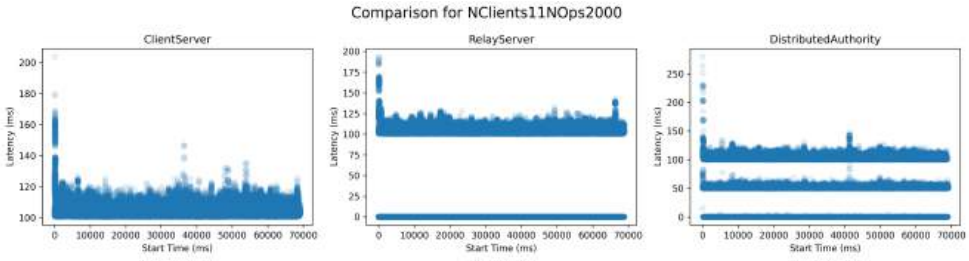

Figure B.25: | Scatter Plot | 11 Clients 2000 Operations

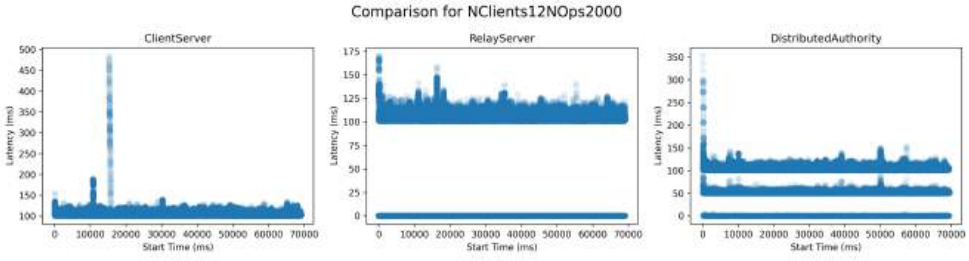

Figure B.26: | Scatter Plot | 12 Clients 2000 Operations

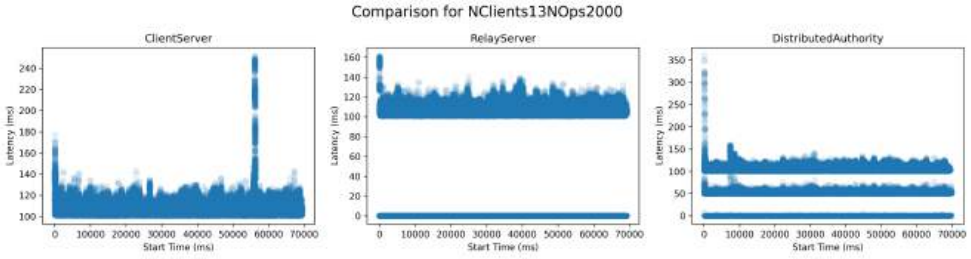

Figure B.27: | Scatter Plot | 13 Clients 2000 Operations

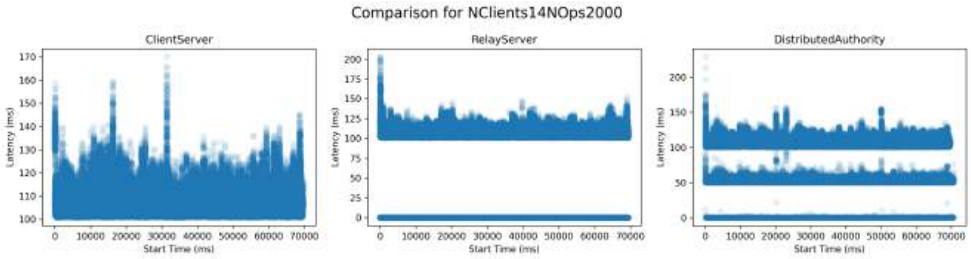

Figure B.28: | Scatter Plot | 14 Clients 2000 Operations

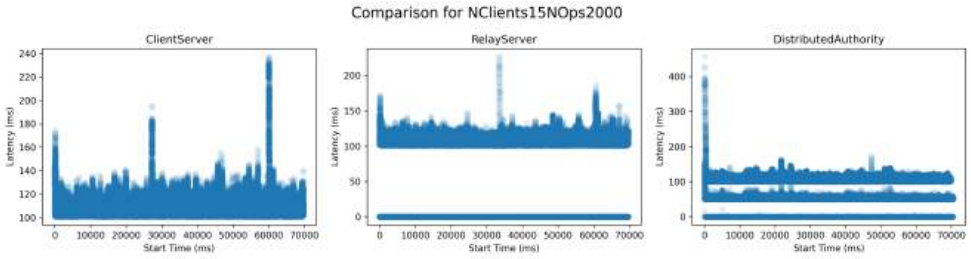

Figure B.29: | Scatter Plot | 15 Clients 2000 Operations

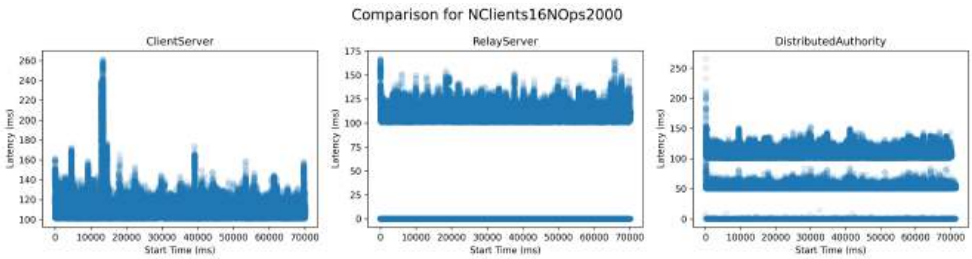

Figure B.30: | Scatter Plot | 16 Clients 2000 Operations

TRITA – EECS-EX 2025:476
Stockholm, Sweden 2025

www.kth.se