
BACHELOR'S THESIS

PeerToCP

A WebRTC-Based Real-Time Collaborative
Code Editor and Shared Shell

Hocky Yudhiono

hocky.yudhiono@gmail.com

SUPERVISOR

Muhammad Hafizhuddin Hilman, Ph.D.

Submitted in partial fulfillment of the requirements for the degree of
Bachelor of Computer Science

Faculty of Computer Science, Universitas Indonesia
Depok, Indonesia, December 2022

English edition. The thesis was written and examined in Indonesian as *PeerToCP: Editor Kode dan Shared Shell Kolaboratif dalam Waktu Nyata Berbasis WebRTC*.

Abstract

Real-time collaborative applications have become part of everyday life, letting people communicate and work together across distance and time. A widely used example is the document editor that several users can edit at once. This thesis presents PeerToCP, a local-first collaborative code editor built on WebRTC and conflict-free replicated data types (CRDTs), together with a shared shell that one user runs and every other user in the network group can use. Three backend variants of the application are compared. For keeping documents consistent, two approaches are evaluated: operational transformation (OT) and CRDTs. For the network architecture, the study evaluates a client-server CRDT variant, a peer-to-peer CRDT variant and a client-server OT variant. A peer-to-peer OT variant was not evaluated, because the OT implementation used here needs a server as its single source of truth. The peer-to-peer CRDT variant performed best for groups of up to eight users ($n \leq 8$), and its signaling server uses minimal resources, so one server can host more network groups. The client-server CRDT variant, however, grows more slowly with the number of users in a group and uses fewer resources on the clients, at the cost of more on the server. It is therefore worth considering when a peer-to-peer network cannot be established, or when groups are much larger than those tested in this study.

Keywords: WebRTC, WebSocket, local-first, real-time, code editor, conflict-free replicated data types, operational transformation, shared shell, Yjs, Electron

Acknowledgements

I thank God Almighty for the grace that allowed me to complete this thesis, *PeerToCP: A WebRTC-Based Real-Time Collaborative Code Editor and Shared Shell*, submitted in partial fulfillment of the requirements for the degree of Bachelor of Computer Science at the Faculty of Computer Science, Universitas Indonesia. I would also like to thank:

- Muhammad Hafizhuddin Hilman, Ph.D., my thesis supervisor, for the weekly supervision and everything I learned from it;
- Dr. Putu Wuri Handayani, Dipta Tanaya, M.Kom., and Dr. Eng. Laksmi Rahadiani, who taught me research methodology and scientific writing;
- all the lecturers who patiently taught me during my studies at the Faculty of Computer Science, Universitas Indonesia;
- my parents, my family and my older siblings, who supported me through my studies while I completed this thesis;
- and my friends: Eko, Joni, Steven Wiryadinata, Samuel, Prabowo, Pikatan, Kenta, Andre, Irfancen, Raihan, Rafi, Aimar, Lucky, Fausta, Novaryo, Rey, and many others I cannot name one by one, for their company and moral support.

I am aware that this thesis still has errors and shortcomings. I hope it proves useful, and an inspiration for the development of science, technology and informatics, in the world and especially in Indonesia.

Depok, 1 December 2022

Hocky Yudhiono

About this edition

This English edition was translated from the Indonesian original with the assistance of Claude, an AI model by Anthropic, and checked by the author. The study, its experiments and its results are unchanged. The edition corrects a few factual slips in the background chapters and in the description of the experimental setup, checked against the published code and data. Every benchmark chart was redrawn from the raw measurements with the same processing as the original, summary charts and explanatory diagrams were added, figures that contained Indonesian text were redrawn in English, and the institutional approval and declaration pages of the original were omitted.

This edition is licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0, <https://creativecommons.org/licenses/by/4.0/>). Its sources, and the script that draws its charts from the raw data, are at <https://github.com/hockyy/peertocp-thesis>.

Contents

Abstract	i
Acknowledgements	ii
1 Introduction	1
1.1 Background	1
1.2 Research Questions.	2
1.3 Scope and Limitations	3
1.4 Objectives and Contributions.	3
1.5 Thesis Outline	3
2 Literature Review	5
2.1 WebSocket	5
2.2 WebRTC	6
2.3 Collaborative Code Editors	8
2.4 Operational Transformation (OT)	8
2.5 Conflict-Free Replicated Data Types (CRDTs)	9
2.6 A Closer Comparison of CRDT and OT.	11
2.7 Related Work	11
3 Research Methodology	13
3.1 Approach and Stages.	13
3.2 Evaluation Method and Scenarios	14
4 Design and Implementation	15
4.1 Libraries and Frameworks.	15
4.1.1 Main Framework and Code Editor Component.	15
4.1.2 Collaboration Components.	16
4.1.3 Shell Component	16
4.2 System Design.	17
4.3 Peer-to-Peer and Client-Server CRDT Variants	19
4.4 Client-Server Operational Transformation Variant.	21
4.5 Evaluation Design	22
5 Results and Discussion	25
5.1 The Local-First and Correctness Aspects.	25
5.2 The Scalability and Responsiveness Aspects.	27
5.3 The Lightweight Aspect	32

6	Conclusion	38
6.1	Conclusions	38
6.2	Future Work	39
	References	40
A	Code and Experimental Setup	44

List of Figures

2.1	Setting up a WebRTC connection.	7
2.2	Styling changes in a rich text editor	8
2.3	TP1 in action	9
2.4	The same insertion in OT and in a CRDT.	10
3.1	Stages of the study.	13
4.1	The components of a PeerToCP client	15
4.2	PeerToCP, peer-to-peer CRDT variant.	17
4.3	How the application is used: a high-level activity diagram.	18
4.4	Running a program on another user's machine	19
4.5	The peer-to-peer CRDT variant.	20
4.6	The client-server CRDT variant.	20
4.7	The client-server OT variant	21
4.8	Synchronization in the OT variant	22
5.1	Incoming network data of the first and second clients, Scenario 1, $n = 8$	26
5.2	Outgoing network data of the first and second clients, Scenario 1, $n = 8$, on a logarithmic scale	26
5.3	Incoming network data of the servers, Scenario 1, $n = 8$, on a logarithmic scale	27
5.4	Mean latency against the number of users n	28
5.5	Time to apply local operations in the code editor, Scenario 3, $n = 8$	29
5.6	Latency of nonlocal operations in the code editor, Scenario 3, $n = 8$	30
5.7	Updates piling up in the OT variant	30
5.8	Time to apply local operations in the shared shell, Scenario 4, $n = 8$	31
5.9	Latency of nonlocal operations in the shared shell, Scenario 4, $n = 8$	32
5.10	Mean resource usage of the client application in Scenario 1 against the number of users n	33
5.11	CPU utilization of the first and second clients, Scenario 1, $n = 8$	34
5.12	Memory growth of the first and second clients, Scenario 1, $n = 8$	34
5.13	Mean resource usage of the servers in Scenario 1 against the number of users n	36
5.14	Synchronization in the peer-to-peer CRDT variant	36
5.15	Connections in a peer-to-peer full mesh (left) and in a client-server star (right), for eight users.	37

List of Tables

4.1	Test scenarios.	23
5.1	Latency of nonlocal operations in Scenario 3 (shared code editor), in milliseconds.. . . .	28
5.2	Latency of nonlocal operations in Scenario 4 (shared shell), in milliseconds.. . . .	30
5.3	Mean resource usage of the client application in Scenario 1.	33
5.4	Mean resource usage of the servers in Scenario 1	35

Introduction

This chapter sets out the background of the study: how real-time communication technology developed, how it is used on the internet today, and what stands in the way of its wider use. From that background it develops the idea of a collaborative application built on these technologies, and states the research questions, scope, objectives and contributions of the thesis.

1.1 Background

Applications that connect people in real time, in speech or in writing, are increasingly common. During the COVID-19 pandemic, when this study was carried out, companies and schools moved their activities online and needed communication tools they could rely on. Common examples include Google Workspace products such as Google Docs, Google Slides and Google Meet, as well as WhatsApp, LINE, JetBrains Code With Me, Zoom and many others. Multiplayer video games, too, let players interact directly, with latency low enough to feel instantaneous.

Real-time communication (RTC) refers to telecommunication methods in which several users interact with a relatively low latency, that is, a short delay relative to the users' responses (Arefin, Azad, & Kabir, 2013). RTC became a focus of research and practice after WebSocket was introduced in 2008 (Fette & Melnikov, 2011; Reynolds, 2008). WebSocket lowers latency and, implemented well, makes real-time communication practical.

Google Docs, one of the most widely used document editors, applies RTC over WebSocket; its real-time collaboration features were developed in 2010 (Belomestnykh, 2010). Google Docs uses operational transformation (OT), a technique that supports a wide range of collaborative capabilities (Day-Richter, 2010; Harris, 2010). Research on OT dates back to 1989 (Ellis & Gibbs, 1989). Over the years, several correctness issues in OT were found and resolved one by one, and implementations vary widely, each with its own trade-offs in memory and time. One of the developers of Google Wave, the predecessor of Google Docs, needed about two years to finish implementing OT (Gentle, 2011). Despite the effort, Google Docs became a flagship real-time collaborative text editor built on a client-server architecture, and it is still in use today.

In 2011, WebRTC was introduced as a protocol and application programming interface for two-way, peer-to-peer real-time communication (Dutton et al., 2012). After a *signaling* step, WebRTC lets each user communicate with every other user directly, without going through a server. Signaling is the process of initiating a WebRTC data channel connection through a server (Sredojev, Samardzija, & Posarac, 2015). WebRTC was designed mainly for the real-time exchange of larger data, such as audio and video (Dutton et al., 2012). Because the server is used only for signaling, its load is also lighter. Compared with a client-server architecture, this tends to improve scalability and network availability for users.

WebRTC has also been studied for other uses, among them real-time collaborative document editing. OT, generally used in client-server architectures, relies on particular properties to make its results converge, which makes it harder to implement in a peer-to-peer architecture (C. Sun, Xu, & Ng, 2017). As its name suggests, OT resolves conflicts by transforming the editing operations applied to a document (Smith, 2012). In a client-server architecture, it relies on the server as a *single source of truth* that resolves every operation arriving from its clients.

Conflict-free replicated data types (CRDTs) have emerged as an alternative way to resolve conflicts in peer-to-peer architectures. CRDTs were introduced in 2006 and formally defined in 2011 (Shapiro, Preguiça, Baquero, & Zawirski, 2011). They are used in distributed computing and do not require every user to be connected at all times. A CRDT resolves conflicts on the *state* of the document, rather than by transforming its editing operations (Preguiça, 2018). CRDTs can also be used in a client-server architecture, in which the server resolves every conflict and broadcasts the changes to all clients (C. Sun, Sun, Agustina, & Cai, 2019).

Both architectures, client-server and peer-to-peer, have advantages and disadvantages. Concentrating reliability and resources on a server can be limiting, but it is more stable, because the developers provision its performance. A peer-to-peer architecture, on the other hand, reduces the communication overhead between users, who connect directly rather than through a server. Peer-to-peer networks work best when the number of users in a network is small (Leibnitz, Hoßfeld, Wakamiya, & Murata, 2007; Maly, Mischke, Kurtansky, & Stiller, 2003). In a full mesh, where every user is connected to every other user, the number of connections grows as $O(N^2)$; data communication then grows quadratically and becomes inefficient, because the same data is sent separately to every user. Whatever the trade-offs, existing real-time collaborative code editors each adopt the architecture their needs call for: the Atom editor with its Teletype plugin, Brackets and JetBrains Code With Me are peer-to-peer, while codecollab, codeshare and Google Docs are client-server.

A collaborative code editor can be developed further into an environment for collaborative competitive programming. Competitive programming is a sport in which individuals or teams solve computational problems under given time and memory limits. The code edited in competitive programming is usually a single file that can be compiled or run on its own, most often in C, C++, Python or Java. This thesis describes the development of a code editor with compiler access and a collaborative shell, which lets every user in a network program together in real time. It also presents a benchmark analysis of three variants of this application, PeerToCP: client-server operational transformation, a peer-to-peer CRDT, and a client-server CRDT.

1.2 Research Questions

Based on this background, the study sets out to answer the following questions.

1. How can several variants of PeerToCP, an application that provides a real-time collaborative code editor and a shared shell, be implemented?
2. How do the WebRTC-based PeerToCP and its other variants compare in performance, latency, and resource usage on the users' machines and on the server?
3. Based on that analysis and evaluation, what are the advantages of a peer-to-peer PeerToCP, and when should each variant be used?

1.3 Scope and Limitations

The research questions are bounded as follows, to make the scope of the study clear.

- The user interface and user experience of the application are not evaluated with human-computer interaction methods.
- The evaluation assumes unconstrained network connections, and a limited number of users, as is typical in competitive programming.
- PeerToCP builds on existing libraries, modules and frameworks, modified and adapted where needed.
- Minor differences in features and behavior between the variants, which do not significantly affect the operations evaluated, are ignored.

1.4 Objectives and Contributions

Within this scope, the study aims to realize the potential of two real-time communication technologies, WebSocket and WebRTC, in PeerToCP, a real-time collaborative competitive programming environment. The implementation is described in detail, including the obstacles met during development and the solutions chosen. The study also compares the implementation variants of PeerToCP through test scenarios that cover the essential operations a user performs. Its contributions are:

- PeerToCP, a desktop collaborative code editor with a shared shell, implemented in three variants: client-server OT, client-server CRDT and peer-to-peer CRDT over WebRTC (Chapter 4; the code is listed in Appendix A);
- a benchmark of the three variants in four test scenarios with groups of two, four and eight users, covering correctness, latency, and CPU, memory and network usage on the clients and the servers (Chapters 3 and 5);
- guidance on when each variant is the better choice (Chapter 6).

More broadly, the study is meant to give an overview and a working prototype of real-time communication technologies such as WebRTC (Web Real-Time Communication), and of methods for synchronizing data replicas among the users of a network. The application could be developed further toward production and commercial use. The study can also serve as a reference for other benchmarks of how architectures perform on particular data-communication operations, especially collaborative text editing and real-time data synchronization. Finally, the development problems and obstacles it reports can be studied further, to find better solutions than those presented here.

1.5 Thesis Outline

The rest of the thesis is organized as follows.

- Chapter 2, Literature Review, covers the technologies and prior work the study builds on: WebSocket, WebRTC, collaborative code editors, operational transformation and CRDTs, and related work.

- Chapter 3, Research Methodology, describes the stages of the study and the aspects on which the system is evaluated.
- Chapter 4, Design and Implementation, describes how PeerToCP and its variants are built, and the design of the evaluation.
- Chapter 5, Results and Discussion, presents and discusses the results of the performance tests.
- Chapter 6, Conclusion, summarizes the findings and suggests directions for future work.

Literature Review

Answering the research questions of Chapter 1 needs some background. This chapter covers the technologies PeerToCP is built on: WebSocket and WebRTC for moving data between users, the properties of collaborative text and code editors, and the two families of algorithms that keep the users' copies of a document consistent, operational transformation (OT) and conflict-free replicated data types (CRDTs). It ends with the CRDT algorithms and libraries closest to this work.

2.1 WebSocket

WebSocket is a communication protocol that provides a two-way, or full-duplex, channel over a single TCP (Transmission Control Protocol) connection (Fette & Melnikov, 2011). The protocol is stateful: the connection between client and server stays open until one side closes it (Pimentel & Nickerson, 2012). In software architecture, WebSocket can be used to build systems with the publish-subscribe pattern (Ganaputra & Pardamean, 2015). A client sends a subscription request to a server over a WebSocket connection, and the server then pushes data to every subscribed client whenever there is an update. Remote procedure calls (RPC) can also run on top of WebSocket. RPC is a distributed-systems mechanism that works like calling a function, with given parameters, on a remote service (Srinivasan, 1995). Over WebSocket, every RPC request reuses the same channel, which is already open, so latency is much lower and there is no per-request setup cost.

WebSocket was introduced in 2008 and standardized in 2011 (Fette & Melnikov, 2011), so by the time of this study it had been in use for well over a decade. Newer protocols are being studied that keep its features while making it faster. Unlike WebSocket, HTTP and HTTPS are stateless, but recent versions of HTTP have moved closer to what WebSocket offers. HTTP/1.1, introduced in 1997 and still in use today, lets the underlying TCP connection persist, so that requests are sent one after another, in order, without opening a new connection each time; the connection is closed once all requests have been answered (Fielding & Reschke, 2015; Krishnamurthy, Mogul, & Kristol, 1999).

HTTP/2, published in 2015, added multiplexing: requests are sent in parallel over a single TCP connection, which makes data transmission more efficient (Belshe, Peon, & Thomson, 2015). In theory, HTTP/2, which provides full-duplex streams, could replace WebSocket (Stenberg, 2014), but its multiplexing and persistent connections were not designed as a replacement for it (Fietze, 2017). HTTP/3 runs over QUIC on top of UDP (User Datagram Protocol) rather than TCP (Bishop, 2022), and it underlies WebTransport, an API and protocol intended as a faster alternative to WebSocket (Frindell, Kinnear, & Vasiliev, 2022). At the time of this study, WebTransport was still an IETF draft, supported only experimentally and only in some browsers (Frindell et al., 2022).

WebSocket connects clients to a server. Other protocols are designed to connect clients directly to each other, in a peer-to-peer architecture; one of them is WebRTC.

2.2 WebRTC

WebRTC is a web technology, available in browsers and on mobile devices, for transmitting data over direct peer-to-peer connections (Alvestrand, 2021a). It is not only an API but also a set of protocols defined by the W3C (World Wide Web Consortium) and the IETF (Internet Engineering Task Force). Google released WebRTC as open-source technology in May 2011, and its API is native to JavaScript (Dutton et al., 2012). Its central component is the connection itself, `RTCPeerConnection`.

An `RTCPeerConnection` represents a connection between a computer and one other peer in a peer-to-peer network (Alvestrand, 2021a; Jennings, Hardie, & Westerlund, 2013; Perkins, Westerlund, & Ott, 2021). In a WebRTC network arranged as a full mesh, each computer in a network of n peers holds $n - 1$ `RTCPeerConnections`, one to each of the other computers. Other topologies reduce the number of connections, each with its own advantages and disadvantages.

WebRTC can exchange continuous media streams, represented by the `MediaStream` component: a stream of audio or video (Alvestrand, 2021a; Sredojev et al., 2015). Alvestrand (2021b), in the IETF document *WebRTC MediaStream Identification in the Session Description Protocol*, describes `MediaStream` in detail. A `MediaStream` contains one or more `MediaStreamTracks`, each an audio or a video track. A track added to an `RTCPeerConnection` is received at the other end of the connection. Media is carried over UDP by default.

The component `PeerToCP` relies on is `RTCDataChannel`, described in the IETF document *WebRTC Data Channels* (Jesup, Loreto, & Tüxen, 2021). A data channel carries arbitrary data, text or binary, over an `RTCPeerConnection`, and a single connection can hold many of them, identified by numbers from 0 to 65534. Data channels run over SCTP (Stream Control Transmission Protocol), which lets each channel choose how its messages are delivered (Jesup et al., 2021). A channel can be ordered and reliable, like TCP, which suits chat messages and files. It can be unordered and unreliable, like UDP, which suits games and remote control: skipping retransmission cuts the overhead of each message, so data arrives sooner. Or it can be partially reliable, with a limit on the number of retransmissions or on how long a message may be retried, and with or without ordering.

Before peers can connect and exchange data, each peer needs to know about the others. A *signaling server* manages the connections between devices, but not the traffic that flows over them (Petit-Huguenin, Nandakumar, Holmberg, Keränen, & Shpount, 2021). It is only an intermediary that tracks the state of a network and which peers are still connected to it. It lets a peer discover the other peers in the network, relays the messages that set up a connection when a new peer joins, and restarts, closes or resets connections when needed.

WebRTC does not prescribe how signaling is done, and there are many alternatives. Signaling can use XMPP (Extensible Messaging and Presence Protocol), plain HTTP requests (XHR), and more (Sredojev et al., 2015). Another common choice is SIP (Session Initiation Protocol), which can run over a WebSocket connection between each client and the signaling server (Adeyeye, Makitla, & Fogwill, 2013). Whatever the transport, the peers describe themselves to each other in the Session Description Protocol (SDP) format.

A session description in SDP carries information about a peer: for example, the kinds of media it will send, its codecs, and how it can be reached. Petit-Huguenin et al. (2021) describes how a connection is set up with SDP, in the IETF document *Session Description Protocol (SDP) Offer/Answer Procedures for Interactive Connectivity Establishment (ICE)*. The peer that starts the connection creates an *offer* and sends it through the signaling server to the peer it wants to reach. That peer replies with an *answer*,

also through the signaling server, and the connection can then be established. How each peer can be reached is described by its ICE candidates: the possible routes another peer can take to reach it directly. Candidates can be included in the SDP itself, or sent separately through the signaling server as they are discovered, one at a time, for example as each new candidate is obtained from a STUN server; this is called *trickling*. Figure 2.1 shows the whole exchange.

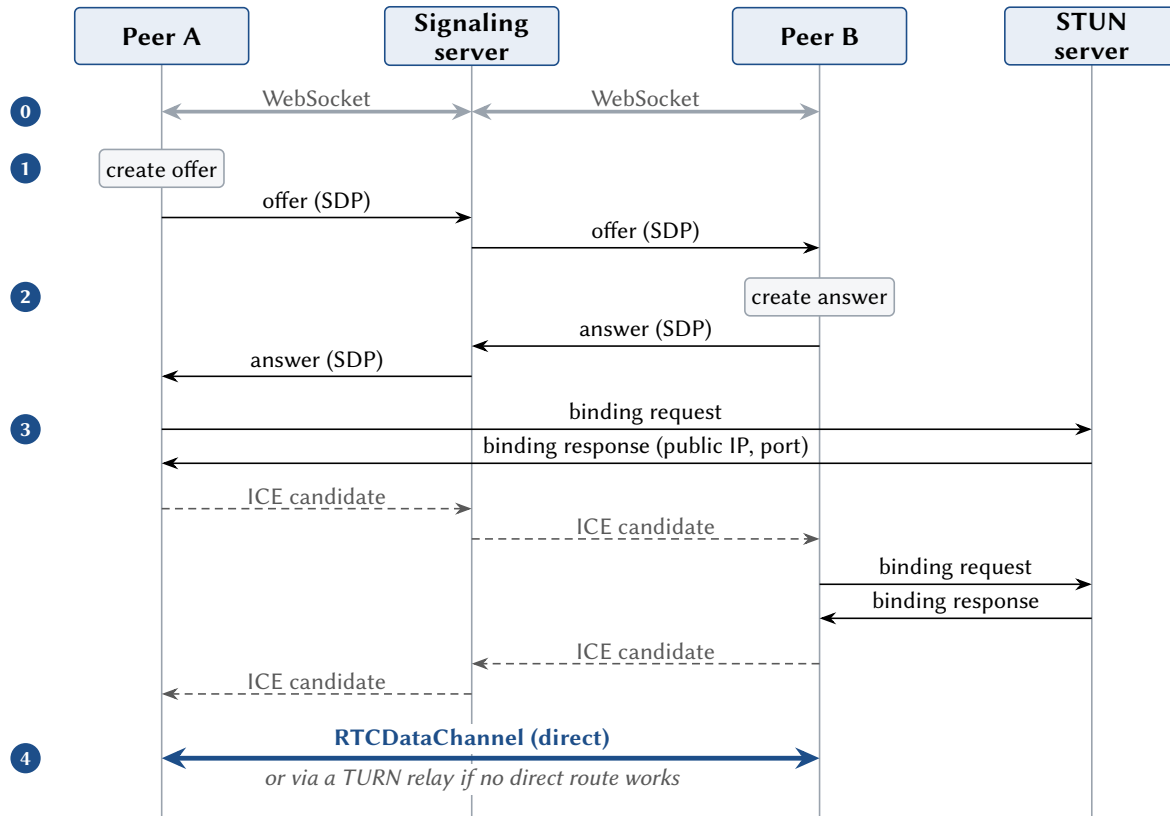


Figure 2.1. Setting up a WebRTC connection. The two peers exchange an SDP offer and answer, and their ICE candidates, through the signaling server; after that, data flows directly between them over an RTCDDataChannel, or through a TURN relay when no direct route works.

Petit-Huguenin et al. (2021) also describes ICE and how it finds addresses. Most of the Internet still uses IPv4, which in practice cannot give every device its own public IP address, so many devices sit behind NAT (Network Address Translation), a mechanism that maps private IP addresses to public ones, and back, as packets travel through the network. NAT is common on small networks, such as a home wireless network or an office. In WebRTC, peers learn how to reach each other through ICE (Interactive Connectivity Establishment). Two kinds of server help a peer gather its ICE candidates: STUN (Session Traversal Utilities for NAT) and TURN (Traversal Using Relays around NAT).

A STUN server tells the peer that contacts it the public IP address and port under which the peer's NAT exposes it. Developers can use free public STUN servers, such as Google's. If a direct connection between the peers fails, for example because a firewall somewhere in the network blocks WebRTC's direct traffic, a TURN server acts as a relay that forwards the traffic. A TURN server has a public IP address that both peers can reach, so it can bridge the two peers (Matthews, Rosenberg, & Mahy, 2010). WebRTC, then, is a complex but flexible set of protocols that suits many uses, and it provides low-latency, peer-to-peer data transmission. It is one of the technologies PeerToCP is built on. The other foundation is an understanding of collaborative code editors, and of the algorithms that make them work.

2.3 Collaborative Code Editors

A code editor is the application programmers use to write their code. Basic features that set a code editor apart from an ordinary text editor include syntax highlighting, automatic indentation and automatic bracket matching (IntelliJ, 2011; Kinder, 2013), and there are many more. For this study, however, every operation in a code editor can be expressed as operations of a plain text editor, in which the characters of the text carry no additional information. Figure 2.2 shows the kind of styling change a rich text editor can make. Operations like this cannot happen in a code editor, because its characters store no additional information.

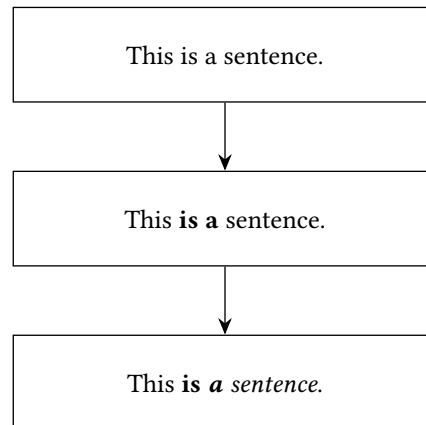


Figure 2.2. Styling changes in a rich text editor: the same sentence with some words set in bold, then some in italic. A code editor has no such operations.

In a collaborative code or text editor, every user holds a replica of the document, and the replicas must end up identical (C. Sun & Ellis, 1998; D. Sun, Xia, Sun, & Chen, 2004). Users are free to edit at the same time. A local operation is applied to the local replica at once, without delay (Attiya et al., 2016; Lv, Cui, & Li, 2015). Each user's operations are propagated to every other user with minimal latency, which makes real-time collaboration possible. An algorithm keeps the replicas consistent: whatever operations the users make, every replica converges to the same document, one that reflects all of those edits (C. Sun & Ellis, 1998; C. Sun et al., 2019; D. Sun, Sun, Ng, & Cai, 2019b; D. Sun et al., 2004). In this sense the operations commute: the algorithm makes the result the same regardless of the order in which the operations reach each replica (C. Sun & Ellis, 1998).

2.4 Operational Transformation (OT)

One challenge in building a distributed system is keeping a piece of data consistent for every client in the system, and the plain text document is one of the data structures research has focused on. Operational transformation (OT) was developed so that every user in a distributed system has the same document after every change (C. Sun & Ellis, 1998). Basic OT has two operations: $\text{insert}(\text{pos}, c)$ inserts the character c at index pos and shifts every character at an index $\geq \text{pos}$ one place to the right, and $\text{delete}(\text{pos})$ deletes the character at index pos (Smith, 2012). These unit operations are the building blocks of OT (Smith, 2012).

OT resolves conflicts between operations whose relative order across clients is not known. It works through a transformation function T , which adjusts the parameters of an operation op about to be

applied to a document to account for the operations that have already been applied to it. An OT algorithm generally has to satisfy two properties (Li & Li, 2004; Martin, 2020; Ressel, Nitsche-Ruhland, & Gunzenhäuser, 1996; Smith, 2012):

- TP1 (Transformation Property 1, also called Convergence Property 1, CP1): $op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2)$.
- TP2 (Transformation Property 2, or CP2): $T(op_3, op_1 \circ T(op_2, op_1)) = T(op_3, op_2 \circ T(op_1, op_2))$.

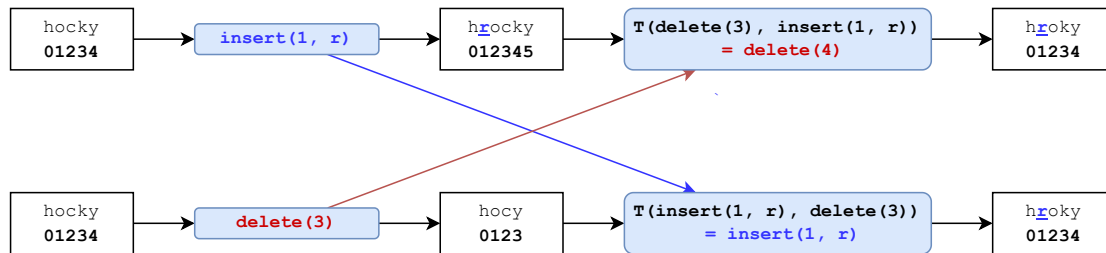


Figure 2.3. TP1 in action. Two sites start from hocky; one inserts r at index 1 while the other deletes index 3. Each site transforms the other’s operation against its own, and both arrive at hroky.

TP1 states that applying op_1 and then op_2 transformed against op_1 gives the same document as applying op_2 and then op_1 transformed against op_2 (Figure 2.3). It matters whenever two operations reach two replicas in different orders. An OT algorithm that satisfies TP1 alone is enough to build a client-server system with a single source of truth. In OT systems that do not rely on TP2, convergence requires every operation to be transformed against one shared context (Xu & Sun, 2016): for example, the history of operations held on the server can be treated as the authoritative order for every client (Vidot, Cart, Ferrié, & Suleiman, 2000). When a client receives new operations from the server, it transforms all of its local operations that the server has not yet accepted against them. The server accepts a client’s operation only once the client already has every operation the server holds.

TP2 states that transforming a third operation op_3 against op_1 and op_2 gives the same result whichever of the two equivalent orders they were applied in. It is needed only in systems that let two operations be executed on two different document states, that is, where concurrent operations are transformed at different sites rather than against one central history. Algorithms that satisfy TP2 are hard to get right: several published ones were later proven wrong (Smith, 2012). This difficulty is one reason CRDTs were introduced as an alternative to OT for keeping data consistent and correct across a distributed system.

2.5 Conflict-Free Replicated Data Types (CRDTs)

In 2005, Oster, Urso, Molli, and Imine (2005) introduced WOOT (WithOut Operational Transformation), the first algorithm that makes the text replicas of a collaborative editor converge without OT. It also preserves the *intention* of an operation, what the operation means to do (Li & Li, 2004). Take the string ABCDE, indexed from 0, and suppose a client inserts the character X between A and B. The intention is to insert X so that A precedes X and X precedes B. In OT this insertion is written $insert(1, X)$; in WOOT it is written $insert(A \prec X \prec B)$, where A and B are objects with an identifier unique to each character in the string (Figure 2.4). WOOT was the starting point for CRDTs, which Shapiro et al. (2011) formally defined in 2011, not only for strings but for many other abstract data types.

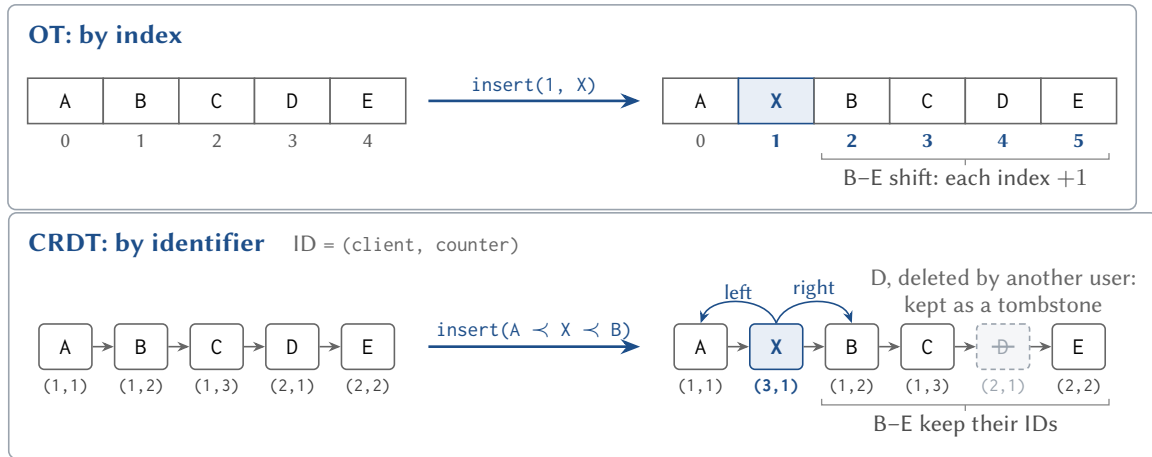


Figure 2.4. The same insertion in OT and in a CRDT. OT addresses characters by index, so a concurrent edit can shift the index and the operation must be transformed. A CRDT gives every character a unique ID and inserts the new character between the IDs of its neighbors; a deleted character stays in the sequence as a tombstone.

A CRDT is an abstract data type that keeps the replicas of a piece of data in agreement across a network. Being an abstract data type means that its semantics are defined by its values and by its operations, so different implementations can realize each operation differently yet behave the same. A CRDT is stored on every client or peer in the network, so memory- and time-efficient implementations matter. Each replica can be modified without coordinating with the others, and if every replica applies the same operations, in any order, they all reach the same final state (Preguiça, Baquero, & Shapiro, 2018; Shapiro et al., 2011).

A simple example is a set. A grow-only set, which only supports $\text{insert}(v)$, is trivially a CRDT, because insertions commute. Supporting $\text{erase}(v)$ as well needs a rule for an insertion and an erasure of the same element that happen concurrently, for example letting the insertion win, as an observed-remove set does (Shapiro et al., 2011). In this study, CRDTs are used for text, so the operations involved are the same as in Section 2.4: $\text{insert}(\text{pos}, c)$ and $\text{delete}(\text{pos})$.

In a CRDT for plain text, every character has its own ID. An insertion carries, along with the new character, the IDs of the characters before and after it, and a logical clock that records the order of insertions, the same idea as in WOOT (Oster et al., 2005). There are two common ways to handle deletion. The first keeps deleted characters in the document forever and only marks them as deleted, a technique known as a *tombstone* (Molli, Urso, Imine, et al., 2006). The second assigns IDs that are already ordered by position: a character inserted at a given position gets an ID greater than that of the character before it and smaller than that of the character after it. These IDs must be able to grow, so that there is always room for a new ID between two others, and each carries a further identifier unique to its client, so that no two IDs are ever the same. Because the order is in the IDs themselves, a deleted character does not need to be kept.

Compared with OT, which needs only its transformation functions and can represent the document as a plain array of characters, a CRDT needs an additional data structure for every character in the document. Each character gets an ID and becomes an object, linked, implicitly or explicitly, to the characters before and after it, as in the two approaches above.

2.6 A Closer Comparison of CRDT and OT

Real-time collaborative text editors in industry are still dominated by operational transformation, even though CRDTs are claimed to have better complexity and efficiency (D. Sun et al., 2019b). The line between the two is not sharp: at heart, both transform operations so that they commute, and they differ in the parameters they use. A CRDT converts an operation into a representation based on IDs, or other identifiers, and back into a position once it is applied; OT expresses an operation directly by the position of its characters.

OT centers on transformation algorithms that manage the concurrent operations of every client. CRDTs center on the content: the ordering of objects, identifier-based operations, and other representations of edits (C. Sun et al., 2019; D. Sun, Sun, Ng, & Cai, 2019a). Proving a CRDT correct is harder than proving an OT algorithm correct, because of the special treatment of its content state (D. Sun et al., 2019a). Every CRDT needs its own criteria to meet its consistency conditions, depending on how it stores its data, and CRDTs for different abstract data types, such as sets, maps and text, each have their own implementation, algorithm and representation. This variety makes convergence harder to prove for CRDTs than for OT, whose basic conditions were proven by earlier research, so that a new OT algorithm only needs to satisfy them (Oster et al., 2005; Smith, 2012; D. Sun et al., 2004).

The two approaches also differ in complexity. In OT, the cost depends on the number of concurrent operations, and nothing is spent on representing the characters of the text as objects. In a CRDT, the cost grows with the size of the document, and setting up a new peer that joins the network also tends to take more time and memory than in OT.

According to D. Sun et al. (2019a), a state-of-the-art OT system takes $O(c)$ time to apply a remote operation and $O(1)$ to apply a local one, with a memory complexity of $O(c)$ or $O(c \cdot m)$, where c is the number of concurrent operations to be transformed and m is the number of users in the network. In practice c is very small ($c \leq 10$), because operations happen in real time, and m is usually at most 5.

In the same analysis, define C as the size of the content without tombstones, and C_t as the content over the document's whole lifetime, tombstones included. The time complexity of a CRDT operation ranges from $O(C_t^2)$ and $O(C_t)$ down to $O(1)$ for tombstone-based CRDTs, and is $O(\log C)$ for solutions without tombstones. Memory can be brought down to $O(C)$, both without tombstones and with tombstones plus garbage collection. For typical documents, $10^3 \leq C \leq 10^6$.

2.7 Related Work

This study uses the Yjs library, which implements the YATA (Yet Another Transformation Approach) CRDT (Nicolaescu, Jahns, Derntl, & Klamma, 2016). YATA represents a document as a linked list of objects and uses tombstones, with a garbage collector that cleans up the content of objects marked as deleted. An insertion adds an object `o(id, left, right, isDeleted, content)` to the list. The `id` is a pair of the peer's ID and a logical timestamp, the number of operations that peer has performed so far. The fields `left` and `right` hold the ids of the objects to the left and to the right of the insertion at the time it was made. The boolean `isDeleted` records whether the object has been deleted, so the editor knows whether to show it, and `content` holds the object's data.

Yjs's main optimization is in content, which can hold a whole run of characters rather than one. When an insertion falls inside the content of an existing object, that object is split in two, so each object also stores

the length of its content. A deletion simply marks the object as deleted. [Nicolaescu et al. \(2016\)](#) give further details of the implementation and its compression. To synchronize, Yjs peers summarize what they have seen as a *state vector*: the latest logical timestamp received from each peer in the network.

Yjs gained popularity in the few years before this study, and other sequence CRDTs have also been implemented. In Logoot, each object's position identifier combines a position with the ID of the site that created it and that site's logical clock, and a deletion removes the object directly, without leaving a tombstone ([Weiss, Urso, & Molli, 2009](#)). LSEQ refines how such position identifiers are allocated ([Nédelec, Molli, Mostefaoui, & Desmontils, 2013](#)). Position-based CRDTs like these can suffer from *interleaving*: when two users concurrently type text at the same place, their characters can end up interleaved with each other, rather than kept as two separate runs ([Kleppmann, Gomes, Mulligan, & Beresford, 2019](#)). YATA orders concurrent insertions that share the same neighbors with a deterministic rule based on the IDs of the clients that made them ([Nicolaescu et al., 2016](#)).

RGA (Replicated Growable Array) is an alternative to YATA ([Roh, Jeon, Kim, & Lee, 2011](#)). It orders concurrent insertions with a pair of the peer's ID and a logical timestamp greater than any timestamp the peer has seen. In practice, implementations of these algorithms add many optimizations beyond the formal algorithms in their papers, such as garbage collection, compression of the data sent over the network, and the way objects are laid out in memory, and these affect their performance.

Research Methodology

This chapter describes the methodology followed to develop PeerToCP: the research approach, its stages, and the aspects on which the system is evaluated.

3.1 Approach and Stages

This study takes an experimental approach. Quantitative and qualitative data are collected for each variant of PeerToCP; all variants implement their business logic behind the same user interface. The independent variables are the network architecture of PeerToCP and the method it uses to resolve conflicts and synchronize data. The dependent variables are defined by the aspects of the system listed in Section 3.2. Quantitative data come from benchmarking and are analyzed descriptively and inferentially, to compare the performance of the variants. Qualitative data come from an objective description of the system. Figure 3.1 outlines the stages of the study.

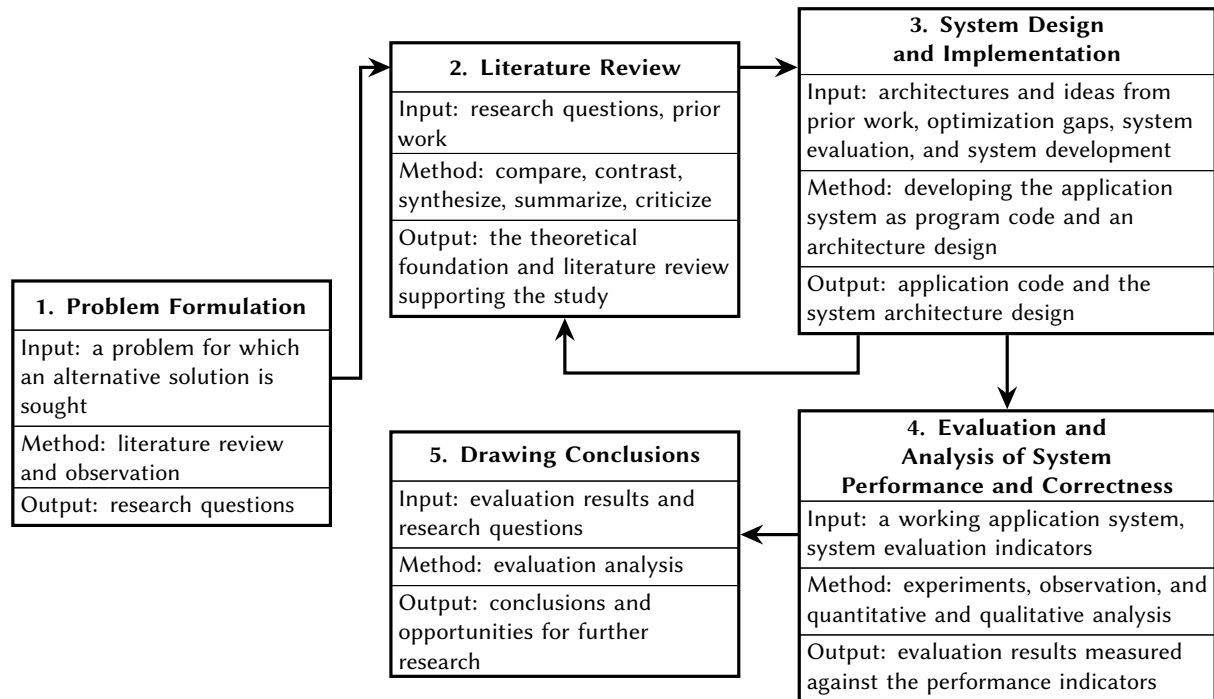


Figure 3.1. Stages of the study.

The research problem was motivated by the potential of WebRTC, a web technology, and of several algorithms for synchronizing data in a distributed network, each with its own advantages and disadvantages. The idea also grew out of a practical need in competitive programming: a simple integrated development environment (IDE) in which code is written collaboratively in real time, and in which a

program run on one client in the network can be used by every other client in it. This problem gave rise to the research questions of the study.

With those questions as input, a literature review covered the relevant technologies and prior work, and produced the theoretical foundation of the study. The review compared related studies in terms of performance, implementation complexity, how they work, and the proofs of correctness of particular technologies and algorithms, to establish how far the technology on this topic had advanced. The study then moved on to designing and implementing PeerToCP. Three variants were tested: operational transformation (OT) on a client-server architecture, conflict-free replicated data types (CRDTs) on a client-server architecture, and CRDTs on a peer-to-peer architecture. The finished system was evaluated objectively on aspects that represent its performance and scalability. These aspects, given in Section 3.2, give a clear basis for comparing one implementation with another.

3.2 Evaluation Method and Scenarios

Based on observations of other real-time collaborative applications, the evaluation covers five essential aspects of each variant of PeerToCP.

1. **Correctness:** every variant of PeerToCP must be correct, meaning that the data replicas kept in sync end up converged and identical.
2. **Lightweight:** the application runs with minimal resources and does not disturb other applications on the same operating system. An application that uses excessive resources to do its job is less likely to be adopted.
3. **Responsiveness:** the data replicas on every client are synchronized with a low, usable latency. Every client of a real-time collaborative application should see minimal delay, so that the experience feels real-time.
4. **Local-first:** an operation is applied to the local replica as soon as the user makes it, without contacting other clients or servers in the network.
5. **Scalability:** this aspect is tied to all the others, since the original motivation for a peer-to-peer architecture is to improve service availability by reducing the load on the server. The application's performance depends on the number of clients or users in a network, so scalability matters for a distributed application ([Leibnitz et al., 2007](#)).

The quantitative and qualitative data from benchmarking are then analyzed, and the conclusions point out opportunities for optimization and for further research.

Design and Implementation

This chapter describes how PeerToCP is designed and built. It introduces the libraries and frameworks it uses and why they were chosen, the design of the system and how it is used, how each of the three variants keeps its replicas consistent, and finally the test scenarios used to compare them.

4.1 Libraries and Frameworks

PeerToCP builds on several third-party libraries, each chosen for a specific reason. The base of the desktop application is Electron. Figure 4.1 shows how the components described in this section fit together in one PeerToCP client.

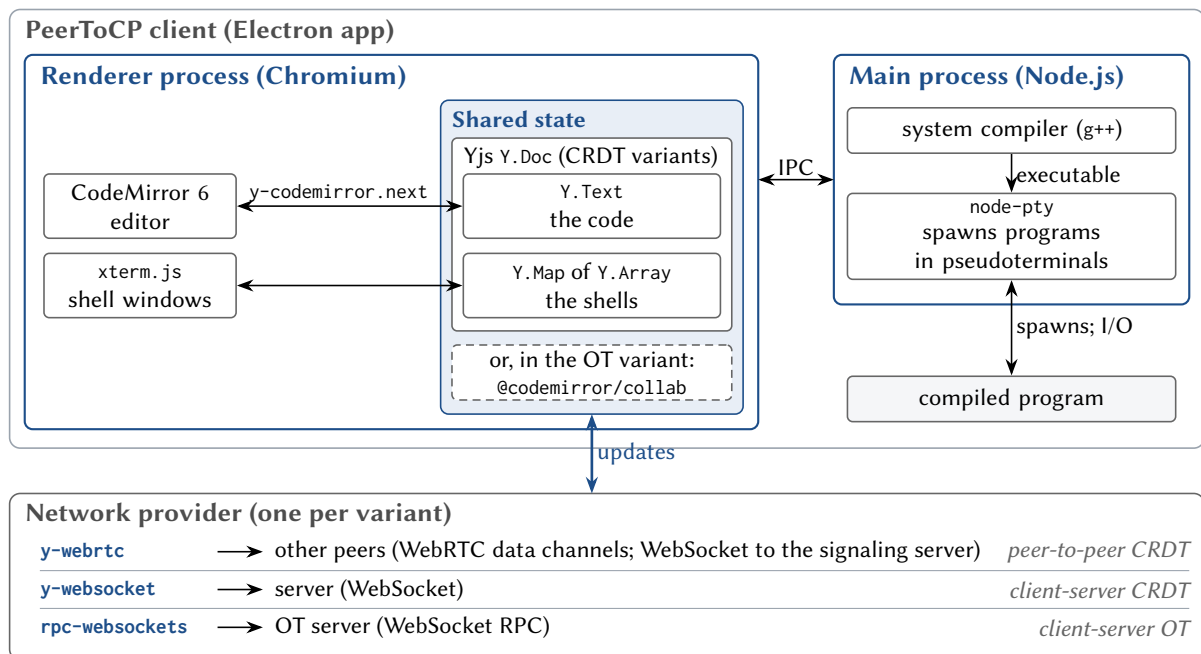


Figure 4.1. The components of a PeerToCP client. The renderer process holds the editor and the shared state; the main process compiles and runs programs; the network provider, which differs between the variants, connects the client to its peers or to the server.

4.1.1 Main Framework and Code Editor Component

Electron is a framework for desktop applications that are built like web applications. Its user interface is rendered by Chromium, from HTML (HyperText Markup Language), CSS (Cascading Style Sheets) and JavaScript, and its back end runs on Node.js (Kredpattanakul & Limpiyakorn, 2018; Miglani & Shriram, n.d.), a JavaScript runtime built on Google’s V8 engine (Tilkov & Vinoski, 2010).

Electron applications are cross-platform: they run on Windows, GNU/Linux and macOS. Electron gives access to many operating system functions, such as spawning subprocesses and writing files. Because its front end is written with web technologies, an Electron application is also comparatively easy to port to the web, with its functionality limited to what the web allows. Alternatives include Qt, based on C++, and Tauri, based on Rust. Electron was chosen because widely used libraries for operational transformation, CRDTs, WebSocket and WebRTC already exist in JavaScript and can be used directly from Node.js.

An Electron application has a main process that runs when the application starts ([OpenJS Foundation, 2022](#)), and each window it opens is a renderer process. The main PeerToCP window is a code editor. Its renderer process can communicate with the main process at any time, and it opens the shell windows in which compiled programs run. The renderer process renders the front end, much like a web page. The central component of the main window is CodeMirror.

CodeMirror is the code editor component, and the user's main point of interaction with the application. It runs in a web browser and models text with an immutable state and modular change operations. It offers many extensions, good accessibility and support for many programming languages. CodeMirror is also the bridge to Yjs, the CRDT library, which already has a CodeMirror binding.

4.1.2 Collaboration Components

Yjs is a library that implements the YATA (Yet Another Transformation Approach) CRDT ([Nicolaescu et al., 2016](#)). Its core is the shared document, `Y.Doc`, which holds the CRDT data structures: `Y.Text` for text, and `Y.Map` and `Y.Array`, which PeerToCP combines to store the shells and their history. This study uses all three, each with its own abstraction.

Yjs is not tied to a particular network architecture. This study uses two network providers that connect a `Y.Doc` to other replicas: `y-webrtc`, for a full-mesh peer-to-peer network, and `y-websocket`, for a client-server network. Yjs has many active developers and was still being maintained and developed, which is why it was chosen for this study.

For the operational transformation variant, PeerToCP uses `@codemirror/collab`, CodeMirror's collaborative editing extension. Its network provider, for the client-server architecture, uses a modified `rpc-websockets` library that supports broadcasting, messaging a specific client, and asynchronous, non-blocking, promise-based remote procedure calls (RPC).

In the OT variant, to keep the program simple, a shell is stored as an append-only array, following OT principles. The operations on a shell are the users' keystrokes. Keystrokes are passed to the terminal as they are, and the program running in the shell decides how to interpret them. A backspace on the shell's input, for example, is recorded by default as the three characters `"\b \b"` (without the quotes): move the cursor left, write a space and move the cursor left again. To collaborate on a shell, every client sends its keystrokes into one shared input channel, without the separate cursors of the code editor.

4.1.3 Shell Component

A shell that runs programs, and that every client in the network can use, needs a library with access to the operating system, to compile and run code. Compilation converts code in a higher-level, human-readable language into binary code a machine can execute ([Aho, Sethi, & Ullman, 1985](#)). Besides editing code

together, every user can compile a single-file program, using a compiler installed on their operating system. In Node.js, the library that gives access to it is `node-pty`.

`node-pty` is a Node.js library that forks processes attached to pseudoterminals. It streams data to and from a process managed by the operating system, and `PeerToCP` uses it to run compiled command-line (CLI) programs, which have no graphical user interface. It was chosen because it is widely used and cross-platform. The shell also needs a front end, for which `PeerToCP` uses `xterm.js`, a component that displays a terminal on a web page. `xterm.js` takes input from the browser and displays output, and it can be connected to a process running on the system. It has no dependencies, which is why it was chosen.

4.2 System Design

`PeerToCP` is a desktop application, which runs directly on the user's computer. This makes offline use easier, since no internet connection is needed to open it. Compiling on a peer also needs access to system calls, which web browser APIs do not provide (Firefox, 2022; Google, 2022): browsers are designed this way so that scripts cannot attack the computer's operating system directly. Figure 4.2 shows the application, and Figure 4.3 outlines how it is used.

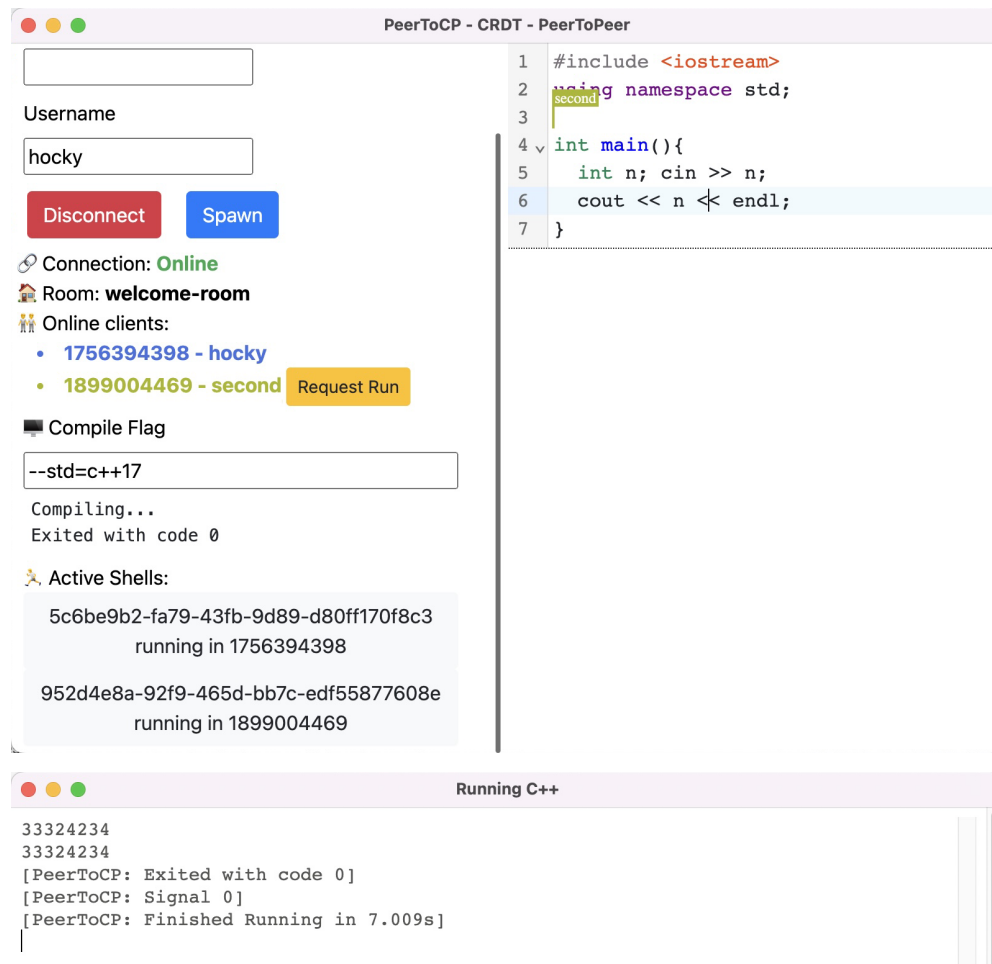


Figure 4.2. `PeerToCP`, peer-to-peer CRDT variant. Top: the main window, with the connection, the room, the online clients (each with a `Request Run` button), the compiler flags and the list of active shells on the left, and the shared code editor on the right, where another user's cursor, labeled *second*, is visible. Bottom: a shared shell window running a compiled C++ program.

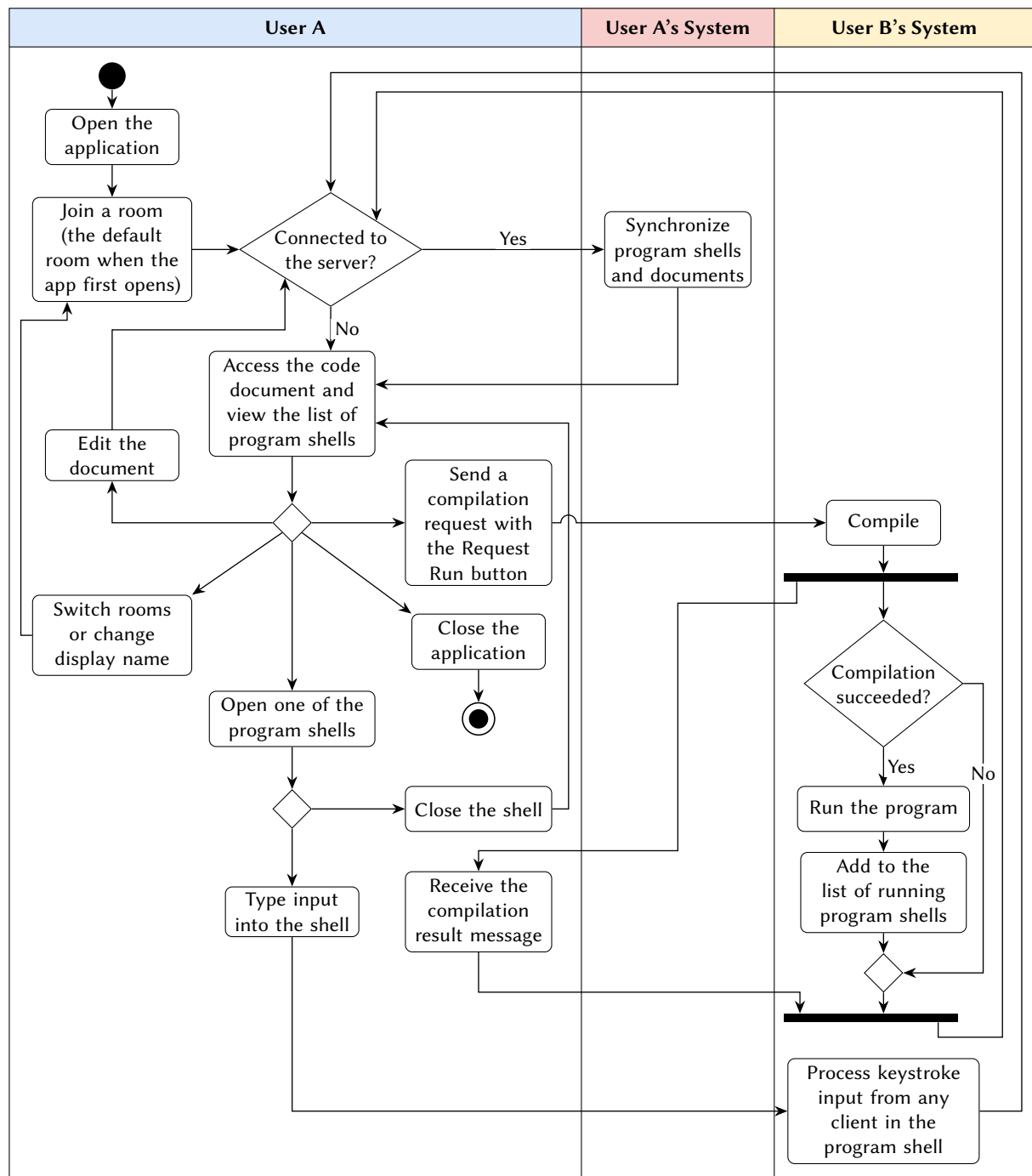


Figure 4.3. How the application is used: a high-level activity diagram.

When users open the application, they join the default room. They can edit the document, and synchronization runs continuously with the publish-subscribe pattern, so updates from other users arrive without having to be checked for, or polled. A user can compile and run the code on their own machine (the *Spawn* button), or send a compilation request to any other user in the network (*Request Run*). PeerToCP delivers the request to the chosen user without involving the other users in the network. The recipient's system compiles the code, runs the program, and adds a shell running it, with its own ID, to the list of shells that every client in the network can access. This list of shells and their contents is stored as a JavaScript object. Figure 4.4 shows this exchange.

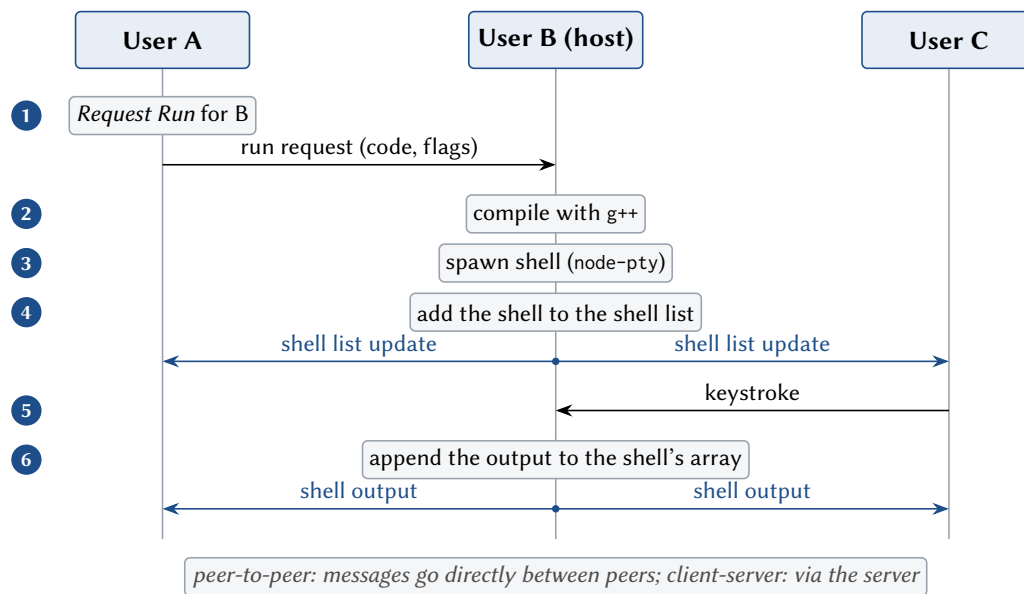


Figure 4.4. Running a program on another user's machine. User A asks user B to run the code; B compiles it, runs it in a new shell and shares that shell with everyone. Keystrokes from any user go to B, the host, and the program's output is replicated to all users. In the peer-to-peer variant the messages travel directly between peers; in the client-server variants they go through the server.

In Figure 4.3, how the program shells and the document are synchronized depends on the variant. In the client-server architecture, user A's application connects to and synchronizes with the server; in the peer-to-peer architecture, it connects to and synchronizes with the other peers directly. Likewise, compilation requests and the messages carrying compilation results go directly between peers in the peer-to-peer architecture, but must go through the server in the client-server architecture.

4.3 Peer-to-Peer and Client-Server CRDT Variants

Both CRDT variants use the Yjs library. Every peer or client holds a replica, a *Y.Doc*, which can contain several nested CRDTs. The code is a *Y.Text*, and the shells are a *Y.Map* that maps the ID of each shell to a *Y.Array* holding its contents. The peer-to-peer variant uses the *y-webrtc* network provider, modified with an interface for sending a message to a specific peer over an *RTCDataChannel* (Figure 4.5). A keystroke typed by any peer is sent directly to the host peer running the program, which appends the program's response to the array of that shell.

The two CRDT variants differ only in their network provider. The client-server variant (Figure 4.6) uses *y-websocket*, also modified so that it can send a message to a specific client through the server. A network provider gives every peer or client in a distributed network an abstraction for communicating

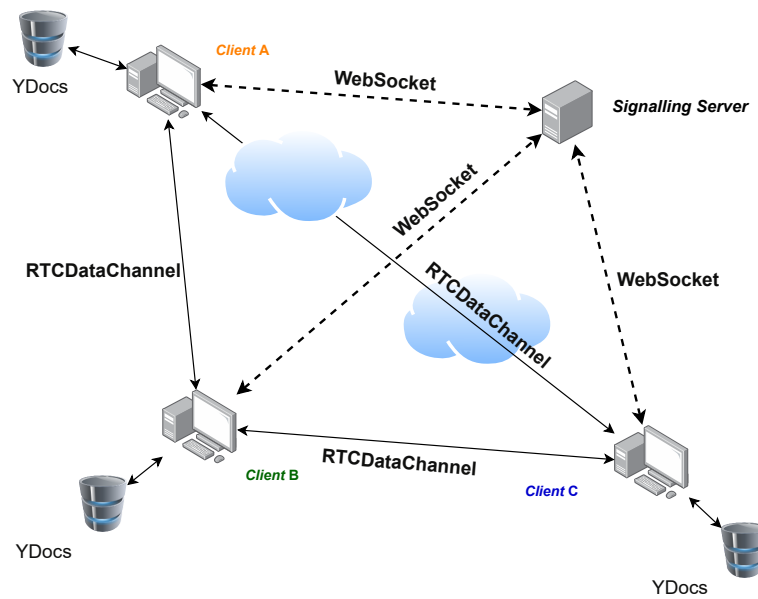


Figure 4.5. The peer-to-peer CRDT variant: every client keeps a Yjs replica and is connected to every other client over WebRTC data channels; a WebSocket signaling server only introduces the peers to each other.

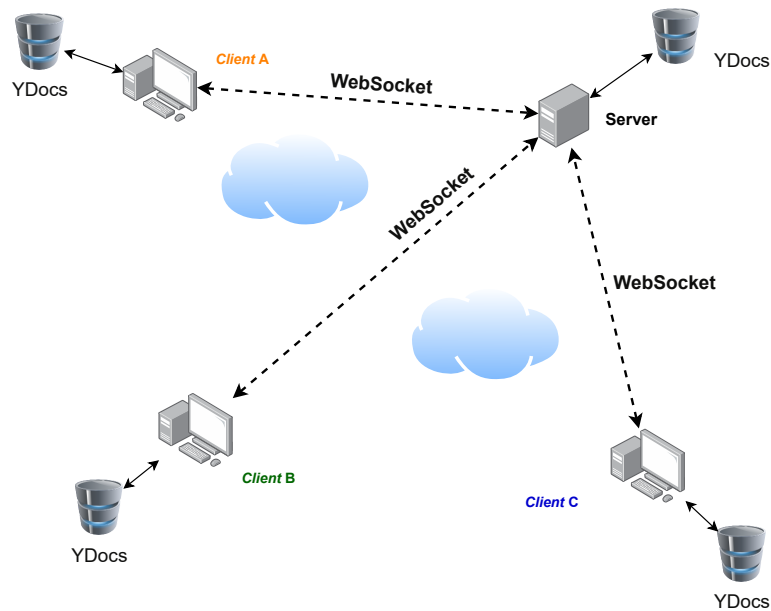


Figure 4.6. The client-server CRDT variant: every client keeps a Yjs replica and synchronizes it over WebSocket with the server, which keeps a replica of its own.

with the others. In the client-server architecture, the server adds one feature: persistence. The server keeps its own replica of the document, so the document survives even when no client is connected. Besides CRDTs, the client-server architecture is also used with operational transformation, in the third variant compared in this study, whose OT satisfies only TP1.

4.4 Client-Server Operational Transformation Variant

For a document that holds only plain text, operational transformation that satisfies only TP1 can be implemented simply on a client-server architecture. This variant uses the `@codemirror/collab` library, which implements operational transformation and represents CodeMirror changes as deltas. The client keeps an array of local changes the server does not yet know about. It sends these changes periodically, together with the latest server version it has applied, through RPC over WebSocket; the RPC call does not block the client while it waits for the reply, before the next attempt to send. If the server already has a newer version than the one the changes are based on, it rejects them and tells the client to rebase: to apply the server's newer updates first, before trying again.

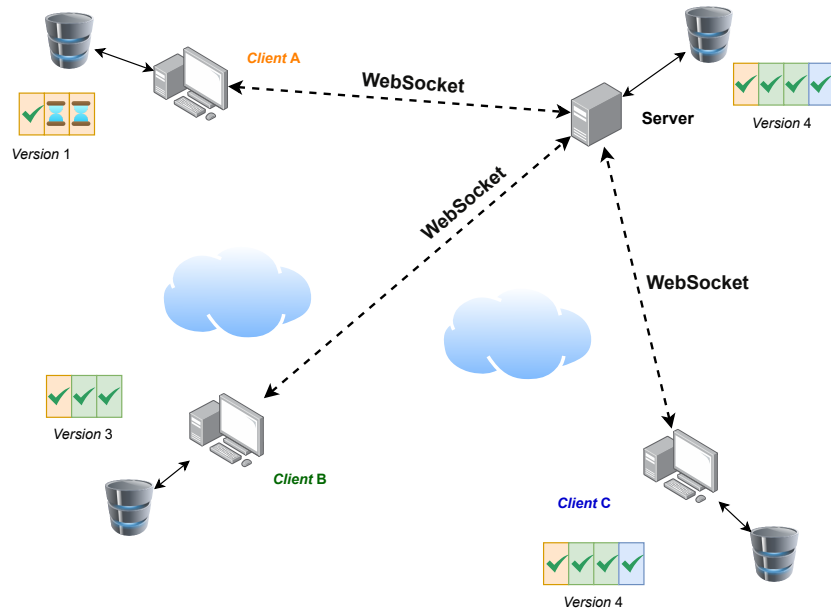


Figure 4.7. The client-server OT variant: the server holds the authoritative sequence of versions, and each client keeps its pending local changes until the server accepts them.

Remote changes received from the server are transformed against the local changes the server does not know about yet; the local changes are then replaced by their versions transformed against the remote ones, and these are what the client sends next. Operational transformation happens only on the clients, and the server applies changes strictly in order, so every replica of the document converges to the same text. Figure 4.7 shows this architecture.

Figure 4.8 shows an example. The server cannot accept client A's two pending updates until client A has all of the server's latest versions. After pulling updates and receiving three new operations from the server, client A transforms its two pending updates against them. Client A's local document is now at version 4, and it can send its updates. The server appends client A's two operations, which brings it to version 6.

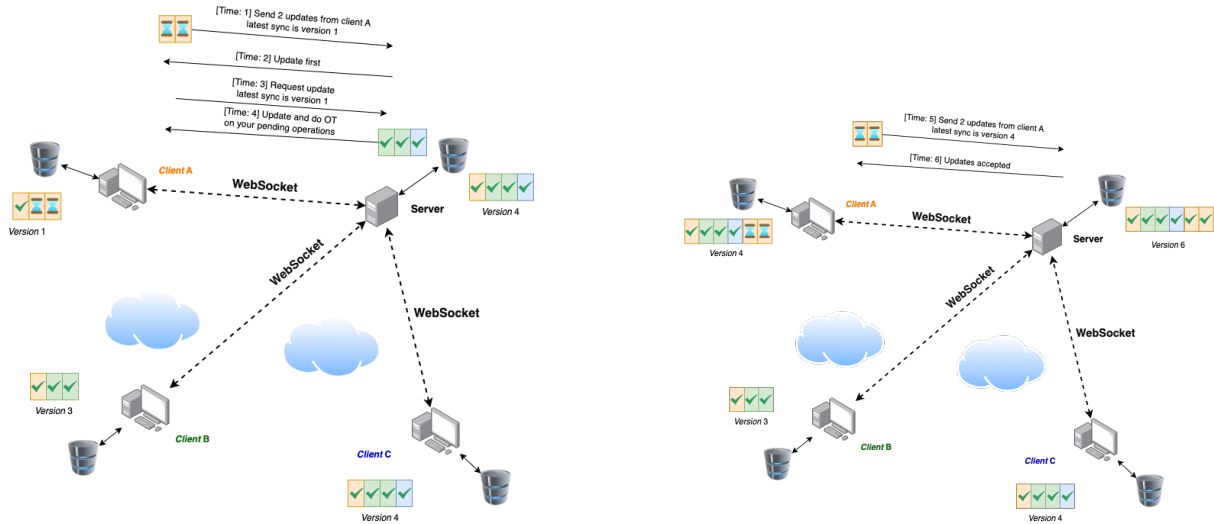


Figure 4.8. Synchronization in the OT variant. Left: client A’s two pending updates are rejected, because the server has moved on. Right: after pulling the server’s new operations and transforming its own against them, client A’s updates are accepted.

The shells of running programs work the same way. When a client types a keystroke into one of the active shells, the keystroke is sent through a similar mechanism, but routed to the client that hosts, that is, runs, the program. The host sends the changes to the shell to the server periodically, as long as there are changes the server has not received, and when a change arrives the server notifies every client to pull it. Because only the host produces the shell’s contents, these updates need no transformation function, unlike plain-text OT. Positions need no mapping either, because the cursor is always after the last character. Applying updates serially in this way prevents race conditions during network disruptions.

4.5 Evaluation Design

The three architectures are back ends for the same editor interface, so the benchmark measures, end to end, what users experience. It consists of scenarios that cover the aspects being tested, run under conditions kept as identical as possible across the variants.

The system is evaluated on the Compute Engine service of Google Cloud Platform. Each variant has its server on an instance in the asia-southeast1-b zone, on an e2-medium machine with 2 vCPUs, 4 GiB of RAM and Debian 11. This machine type was chosen so that the server’s specifications do not limit the benchmark. No relay (TURN) server is provided for peers that cannot connect to each other in the peer-to-peer variant: the study assumes that the WebRTC connections between peers can always be set up.

The users are e2-small virtual machines with 2 vCPUs, 2 GiB of RAM and Debian 11, all in the asia-southeast2-a zone, a region close to the servers, to approximate real-world use of PeerToCP. Four test scenarios are run, each with $n \in \{2, 4, 8\}$ users, to show how the system scales with the number of users in a network. The largest value of n reflects the expected maximum number of users of PeerToCP, which is no more than eight. Table 4.1 summarizes the scenarios.

Table 4.1. Test scenarios.

Scenario	Component	Measures latency	Duration (s)	Disconnection
1	Code editor	No	180	Yes, 30 s
2	Shell	No	≈ 100	Yes, 20 s
3	Code editor	Yes	120	No
4	Shell	Yes	≈ 100	No

Scenario 1: editing under stress. The first scenario simulates a stream of edits in the code editor, with each number of peers n , for the scalability aspect. Every 100 milliseconds, for three minutes, each client performs one of three random operations, chosen with equal probability:

1. inserting (insert) a random string of 2 to 5 characters;
2. deleting (delete) a range of 1 to 3 characters;
3. replacing (replace) a range of 2 to 5 characters with a random string of 2 to 5 characters.

On each client, these operations grow the document by about 5 characters per second on average, and change about 41.67 characters per second, insertions and deletions together. The tests start at the same time on every client, using the operating system's clock, which is synchronized by default. Some error between the clocks is unavoidable and is assumed to be negligible; identical instances for every client reduce the bias from differences between them. In addition, each client disconnects from the network for 30 seconds, starting at a random time between 60 and 90 seconds into the test, so that the disconnection always falls between the first and the last minute. At the end, every editor is connected again, for a final synchronization with the other users. While an editor is disconnected, it keeps performing insert, delete and replace operations, which tests the local-first aspect.

The first scenario is a stress test of the editing that can happen in the real world. After the test, the system waits for one minute, to give synchronization time to finish. The time of the last update is recorded and compared, for the responsiveness and scalability aspects. The hash of the final document is recorded on every client and compared across them, for the correctness aspect. The RAM, CPU and network activity of the first two clients of each run, and of the server, are recorded with the Netdata monitoring agent, for the lightweight and scalability aspects.

Scenario 2: a shared shell under disconnection. The second scenario uses the same environment as the first. It tests the shell with each number of clients n , by running a C++ program that prints 100 random unsigned 64-bit integers, waiting a random 500 to 1500 milliseconds before each one. Meanwhile, a 20-second network disconnection starts at a random time 10 to 40 seconds after the program starts. As in the first scenario, the last synchronization time, the device activity and the hash of every shell on every client are recorded and compared, to make sure that every user ends up with the same shell replica.

Scenarios 3 and 4: latency. The last two scenarios measure the latency of the code editor and of the shell, respectively. In the third scenario, the n peers are set up as before, and each performs two kinds of operation: inserting a timestamp into the editor, and replacing a range of 10 to 15 characters with a timestamp. Each operation waits a random 500 to 1000 milliseconds, so that the operations do

not all happen at a fixed period. Every client captures the timestamps it receives and measures the latency as the time between a timestamp being written and its arrival at another client. In the fourth scenario, every client runs a C++ program that prints 100 timestamps, with a random delay of 500 to 1500 milliseconds between them. Every client captures each timestamp and measures its latency in the same way, as the time between the host of the shell (the user running the program) adding the timestamp and each other user receiving it. Scenarios 3 and 4 have no disconnections, so that the data for the responsiveness aspect are as accurate as possible.

Results and Discussion

This chapter presents and analyzes the results of the evaluation, run according to the scenarios of Section 4.5. The analysis is organized by the aspects defined in Chapter 3, to give a clear picture of how each variant performs and what resources it needs to do so. Along the way, it considers which variant of PeerToCP is preferable and why, and points out weaknesses of the system as directions for future development.

5.1 The Local-First and Correctness Aspects

These two aspects go together. In Scenarios 1 and 2, the local-first aspect was tested by disconnecting clients at random while every user kept changing their local data offline. The changes were indeed local-first: they could still be made, and were applied immediately, while the client was offline, and when the client rejoined the network its data was brought up to date with the changes made in the meantime.

In the experiments, the clients of every variant ended each scenario with the same document hash, that is, in the same final state. The exception is Scenario 1 with eight clients, in which some clients of the client-server OT variant could not reconnect after their disconnection. The likely cause is the inefficient way this variant transmits data, which pushed the bandwidth it needed beyond the limits of the test environment.

Scenario 1 with eight clients was run up to three times for the OT variant, and in each run one or two clients could not reconnect. As described in Section 4.5, every client was disconnected at a random time, for the same duration. Figure 5.1 shows the incoming network data of the first and second clients of each variant in this scenario.

The incoming traffic in Figure 5.1 looks normal: the updates received are not very large. Figure 5.2, however, shows that the first client of the OT variant, which was disconnected at second 72 and reconnected at second 102, then sent a large amount of data for about 20 seconds. The same holds for the second client, disconnected at around second 60 and reconnected at second 90: right after a client reconnects, its outgoing traffic spikes. With many users in one document, disconnections and reconnections at similar times can make this variant unreliable.

An update sent right after the connection is restored tends to be large, because it gathers the operations made while the client was offline. When it is sent, the server is also receiving updates from many other clients. The OT algorithm implemented here accepts an update only if the client's latest version matches the server's; otherwise, the server rejects the update and asks the client to pull first. Meanwhile, the backlog of large updates keeps being resent, and can saturate the server's bandwidth.

The traffic into the server of the client-server OT variant can exceed 800,000 kilobits per second, that is, 100 megabytes per second. Figure 5.3 shows that its mean rate is about 950 times that of the client-server

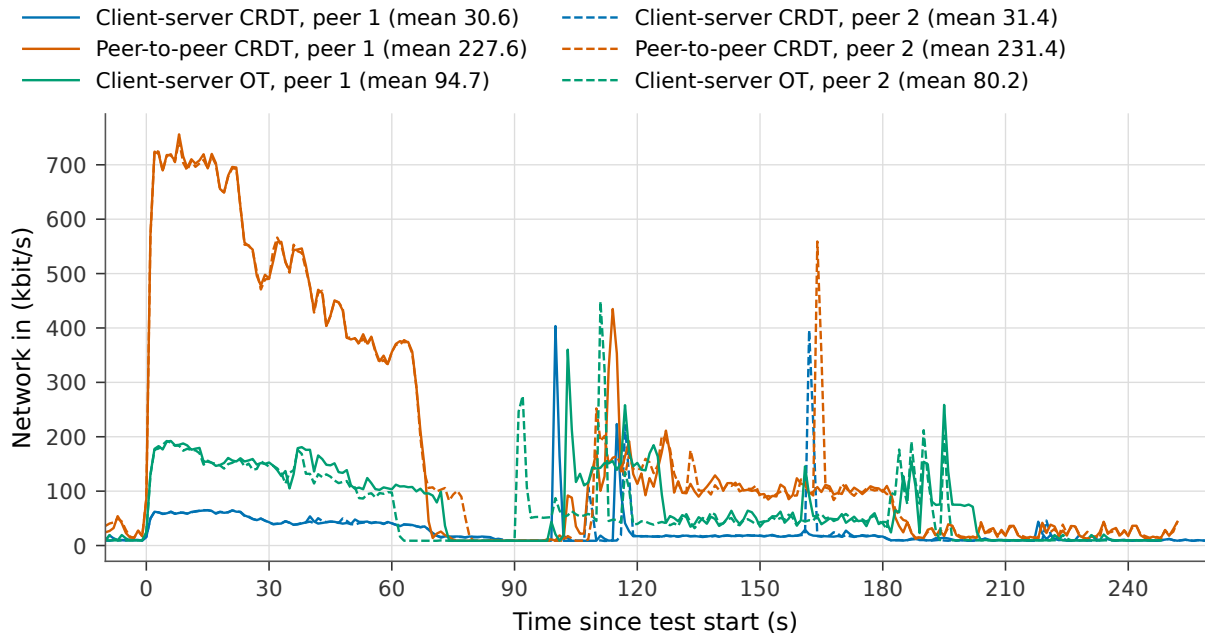


Figure 5.1. Incoming network data of the first and second clients, Scenario 1, $n = 8$.

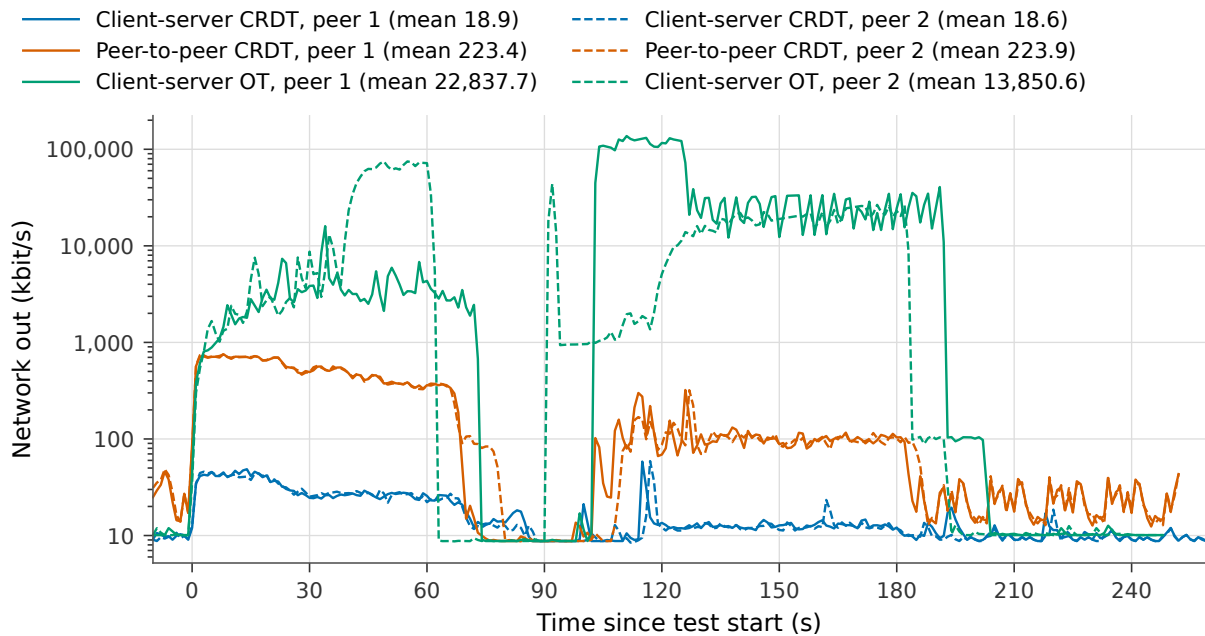


Figure 5.2. Outgoing network data of the first and second clients, Scenario 1, $n = 8$, on a logarithmic scale. The OT clients send up to a thousand times more than the CRDT clients, in bursts after their reconnection.

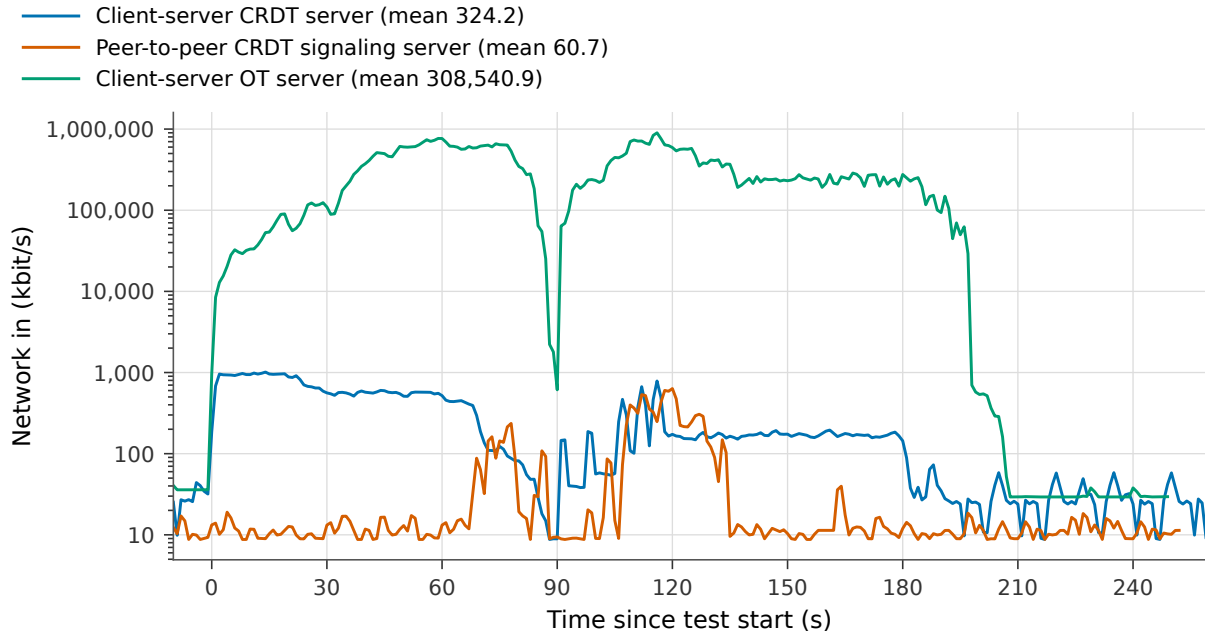


Figure 5.3. Incoming network data of the servers, Scenario 1, $n = 8$, on a logarithmic scale. The signaling server of the peer-to-peer variant stays close to zero.

CRDT variant. The signaling server of the peer-to-peer CRDT variant, by contrast, shows almost no activity: it does not take part in the data exchange, because the peers communicate directly.

In Scenario 1, the CRDT variants are more reliable, because updates to the document can be merged in any order. Other likely factors are the implementation of the WebSocket provider and the compression and encoding Yjs uses. In the OT variant, updates also pile up because of large change operations, which take longer to arrive than the smaller updates of other clients. As a result, some clients have to pull repeatedly before they can push their own update, whenever their local version differs from the server's. This in turn affects two other aspects: scalability and responsiveness.

5.2 The Scalability and Responsiveness Aspects

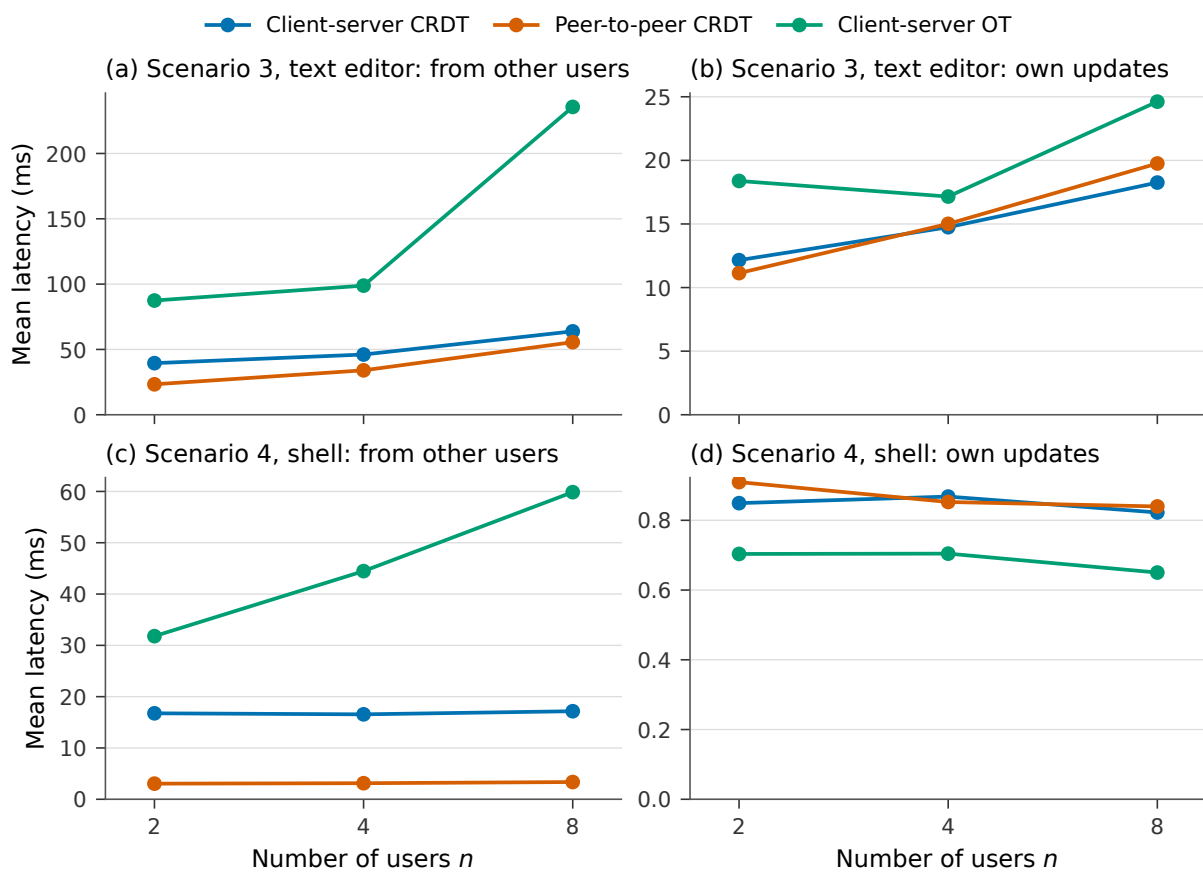
The latency of the code editor was measured in Scenario 3, which, unlike Scenarios 1 and 2, has no disconnections. Each user measured the delay with which data from every other user arrived, recorded when the document data arrived, and these latencies were averaged over the n users in the network. Table 5.1 gives the latency of nonlocal operations, that is, operations made by the other users in the network. The peer-to-peer CRDT variant has the lowest latency, although the gap between it and the client-server CRDT variant narrows as n grows. The latency of the client-server OT variant is more than twice that of the client-server CRDT variant, nearly four times at $n = 8$, with spikes of about two seconds at $n = 8$.

As for scalability: when all users are close to one another (in this experiment, all peers were in the same zone, in Indonesia), the peer-to-peer architecture performs better than the client-server one for the values of n tested. The narrowing gap in Table 5.1 suggests, however, that the client-server CRDT variant may offer better responsiveness for larger n , or when the peers are farther apart.

Figure 5.4 summarizes the latency measurements of Scenarios 3 and 4 against the number of users.

Table 5.1. Latency of nonlocal operations in Scenario 3 (shared code editor), in milliseconds.

n	Peer-to-peer CRDT			Client-server CRDT			Client-server OT		
	Mean	Median	Max	Mean	Median	Max	Mean	Median	Max
2	23.30	21	76	39.50	38	134	87.42	84	168
4	34.00	30	223	46.10	42	220	98.84	86	1,225
8	55.56	47	313	63.84	56	308	235.62	151	2,010

**Figure 5.4.** Mean latency against the number of users n . (a) and (c): updates from other users, as in Tables 5.1 and 5.2; (b) and (d): the time to apply a user's own updates.

Part of the difference in latency may also come from the algorithm that keeps the replicas identical. To isolate it, the time to apply local changes was measured for each variant. A local operation here is one that a user makes on their own document: it is local-first and involves no network transmission. Figure 5.5 shows only a small difference between the variants, about 5 ms on average, and small spikes, which can occur when characters are inserted or deleted at particular random positions. Note that the costs of CRDT and OT scale differently: the time complexity of OT depends on the number of concurrent operations, while that of a CRDT depends on the number of characters, that is, on the size of the document.

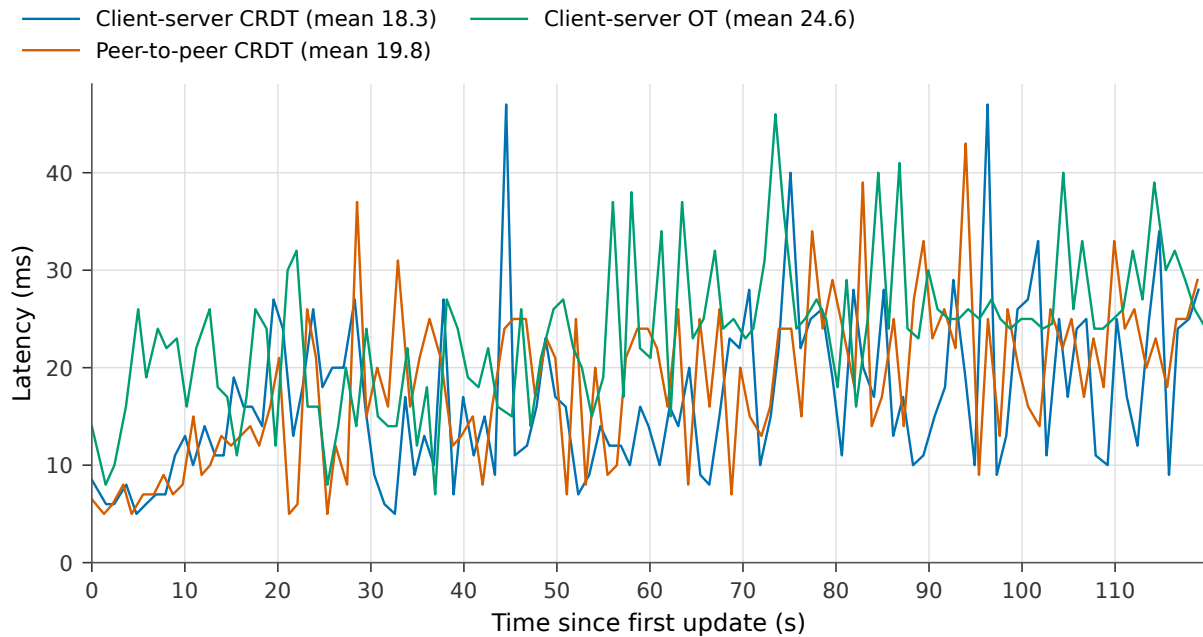


Figure 5.5. Time to apply local operations in the code editor, Scenario 3, $n = 8$.

The total latency is the time to apply local and nonlocal operations plus the network transmission time, which differs between the architectures. Figure 5.6 shows larger differences between the variants, and several latency spikes over the course of Scenario 3. They have two likely causes: the time to resolve nonlocal operations, and the pile-up of updates illustrated in Figure 5.7.

In the client-server OT architecture, the server does not accept an update from a client that does not have its latest version yet. Updates pile up when a client cannot get its data accepted, because its update keeps growing while other clients always get theirs in first. This holds back the updates of one or more clients, and makes the data the clients send, and the server receives, much larger than in the other variants.

In realistic use, this should not happen often, because users do not insert, delete and change characters as fast and as simultaneously as in Scenario 1. Repeated copy-and-paste operations could cause it, since they change many characters at once. Such operations, however, usually come with enough time between them for every client to update its replica before the next large change.

Scenario 4 tests the latency of the shared shell, which every user in the network can use at the same time; Table 5.2 summarizes it. In the code editor, applying a local operation took about 10 to 25 ms on average (Figure 5.5). The shell's data structure is a simple append-only list, so applying an operation takes almost no time in any variant, and the variants show no notable difference (Figure 5.8).

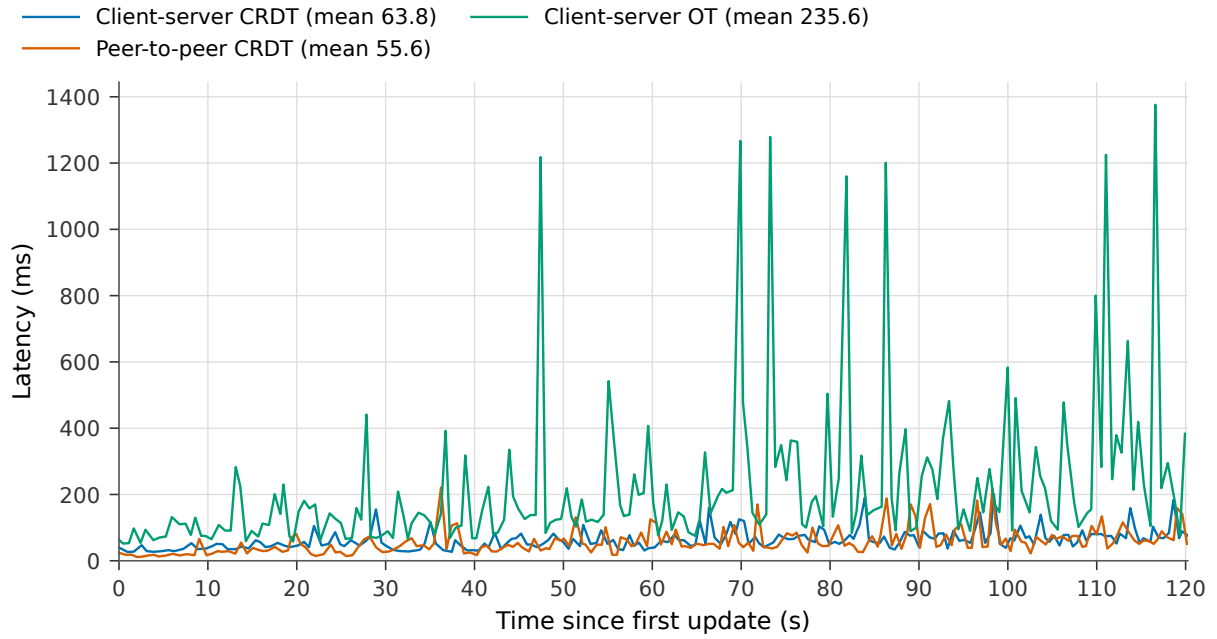


Figure 5.6. Latency of nonlocal operations in the code editor, Scenario 3, $n = 8$.

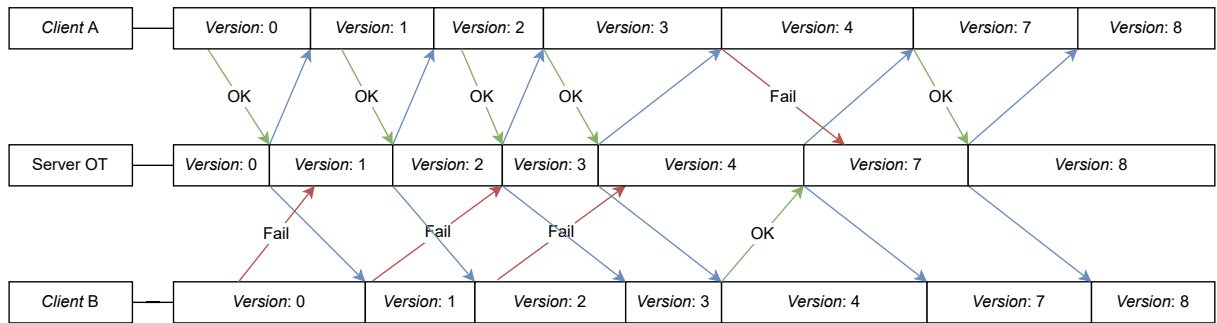


Figure 5.7. Updates piling up in the OT variant. Green arrows are pushes the server accepts, red arrows pushes it rejects because the client is behind, and blue arrows pulls. Client B's pushes keep failing while client A keeps getting its own in first.

Table 5.2. Latency of nonlocal operations in Scenario 4 (shared shell), in milliseconds.

n	Peer-to-peer CRDT			Client-server CRDT			Client-server OT		
	Mean	Median	Max	Mean	Median	Max	Mean	Median	Max
2	3.05	3	5	16.75	17	19	31.79	30	170
4	3.13	3	6	16.55	17	21	44.47	32	309
8	3.36	3	19	17.16	17	35	59.86	30.33	551

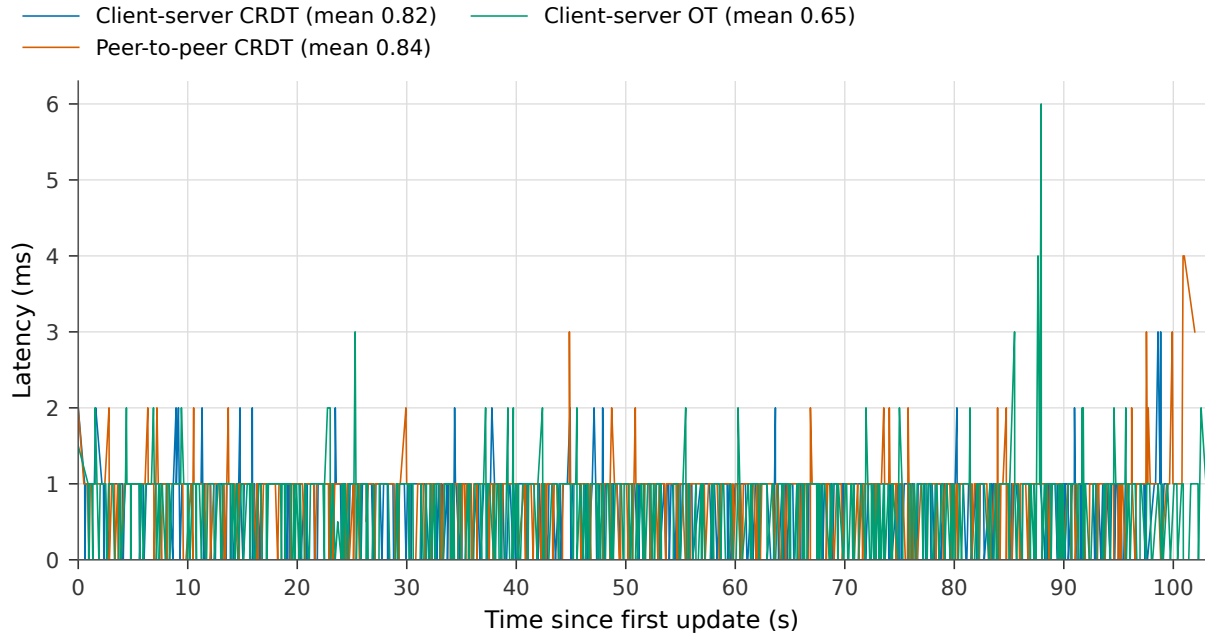


Figure 5.8. Time to apply local operations in the shared shell, Scenario 4, $n = 8$.

This is because appending to an append-only array takes amortized constant time. For the shell, scalability is therefore mainly a matter of how efficiently the data is transmitted, and shell updates in the OT variant suffer from the same problem as in Scenarios 1 and 3, as Figure 5.9 shows.

Another reason why OT is slower than the client-server CRDT is its communication mechanism. The Yjs CRDT implementation encodes and decodes updates in a compact binary form, which makes transmitting large data faster, and a CRDT does not need updates to be applied in order. The OT variant, in contrast, uses the same update mechanism for the shell as for its code editor.

In this experiment, the peers of the peer-to-peer variant were close to one another, in the same zone, and their latency was lower than in both client-server variants, whose traffic must pass through the server first. For $n \leq 8$, the peer-to-peer variant gives the best responsiveness of all the variants. For $n > 8$, or for peers farther apart, the client-server CRDT variant is expected to be worth considering as the better alternative.

Peers in the WebRTC-based variant may also be unable to connect to one another at all. A TURN relay server can then forward their traffic, at a latency that further experiments would have to measure. A better OT algorithm is also worth considering: in theory, its constant-time append operations could give the shell lower latency, by exploiting the append-only array and the fact that only the host of a shell forwards its interaction data. The CRDT data structure for the shell could likewise be modified to exploit these properties.

The system could also choose the best architecture automatically: peer-to-peer without a relay server, as in this experiment; peer-to-peer with a relay server, for peers that cannot connect directly over WebRTC; or client-server, when a network group has many clients. For latency, this choice could rest on a simple metric such as the ping between users, measured in the background. Other factors must be weighed as well, above all the lightweight aspect: how much processing, bandwidth and memory the server and the users must provide.

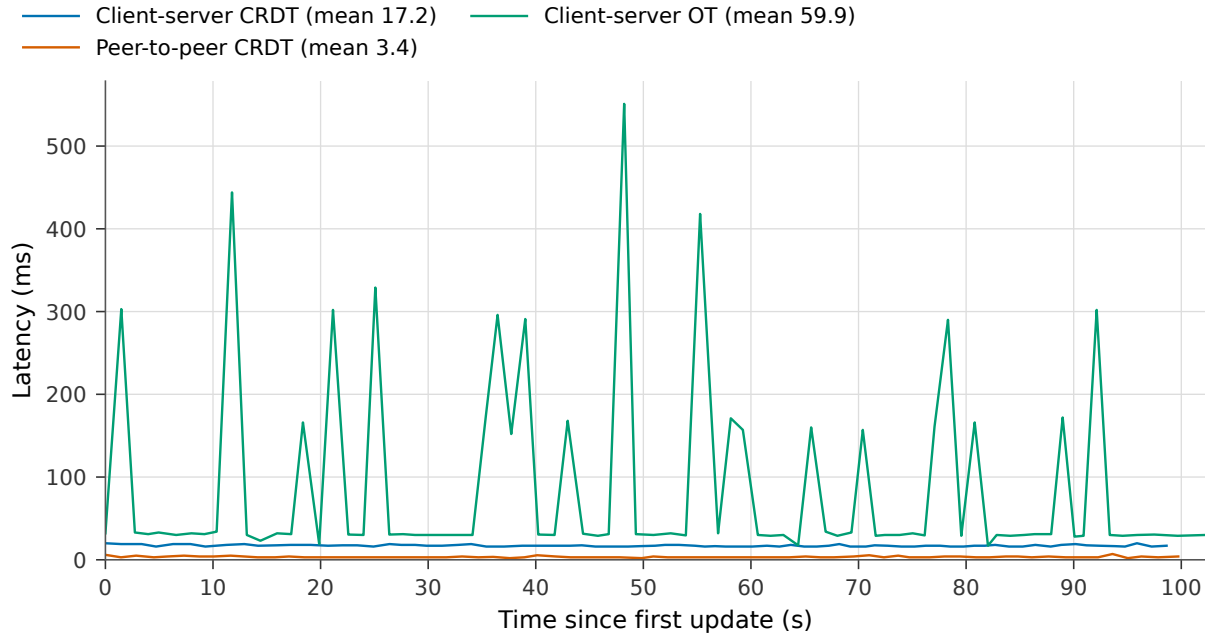


Figure 5.9. Latency of nonlocal operations in the shared shell, Scenario 4, $n = 8$.

5.3 The Lightweight Aspect

This aspect describes how much computing resources each user and the server need. All RAM measurements were normalized to zero just before each test started, so that their growth can be compared. On the clients, the Electron application runs on Chromium, which has its own garbage collection and caching (Thompson, 2015), so memory is more meaningfully measured relative to its size just before the test. At the start, each server uses relatively little RAM, under 20 MiB, while each client starts at about 200 MiB, because the Electron framework runs a Chromium web browser for its front end.

In Table 5.3, 100% CPU utilization means that the application fully uses one vCPU core. RAM usage and incoming data of the peer-to-peer CRDT variant grow the most with n (Figure 5.10): the more peers there are in the network, the more replicas each peer must keep in sync, because the signaling server of this variant plays no part in transmitting CRDT data, and every peer exchanges updates with every other peer itself. All the data a CRDT client receives can be used immediately, with no rejected updates as in the OT variant. In terms of outgoing data, the CRDT variants therefore meet the lightweight aspect much better than the OT variant.

Overall, the client-server CRDT variant is the lightest on the clients. In terms of CPU, on test machines with two vCPU cores (a maximum utilization of 200%), every variant in Scenario 1 with $n \geq 4$ uses on average only about a third of the CPU, as Figure 5.11 shows. These numbers should be read with the caveat that Scenario 1 is a much heavier load than normal use is expected to be. The operating system also decides which processes to prioritize, in memory and in CPU, while the application runs, so resource usage may be higher or lower depending on other activity on the machine.

Before normalization, memory usage starts at about 200 MiB on average, and then grows as Figure 5.12 shows. Both CRDT variants grow less than the OT variant. Every variant, however, uses little memory compared with the capacity of the test machine, about 15%. As long as the application does not load the

Table 5.3. Mean resource usage of the client application in Scenario 1. CPU in percent of one vCPU, RAM growth in MiB, network rates in kbit/s. Each value is the mean of the first two clients.

Variant	Metric	$n = 2$	$n = 4$	$n = 8$
Peer-to-peer CRDT	CPU	39.20	60.14	62.29
	RAM	16.58	19.24	21.30
	Net in	28.52	96.17	229.48
	Net out	28.04	93.98	223.63
Client-server CRDT	CPU	33.96	57.39	56.89
	RAM	14.93	19.13	20.95
	Net in	19.47	27.69	30.97
	Net out	19.76	21.21	18.73
Client-server OT	CPU	47.79	73.29	65.26
	RAM	25.95	31.17	36.34
	Net in	62.24	99.81	87.42
	Net out	12,962.05	28,872.47	18,344.15

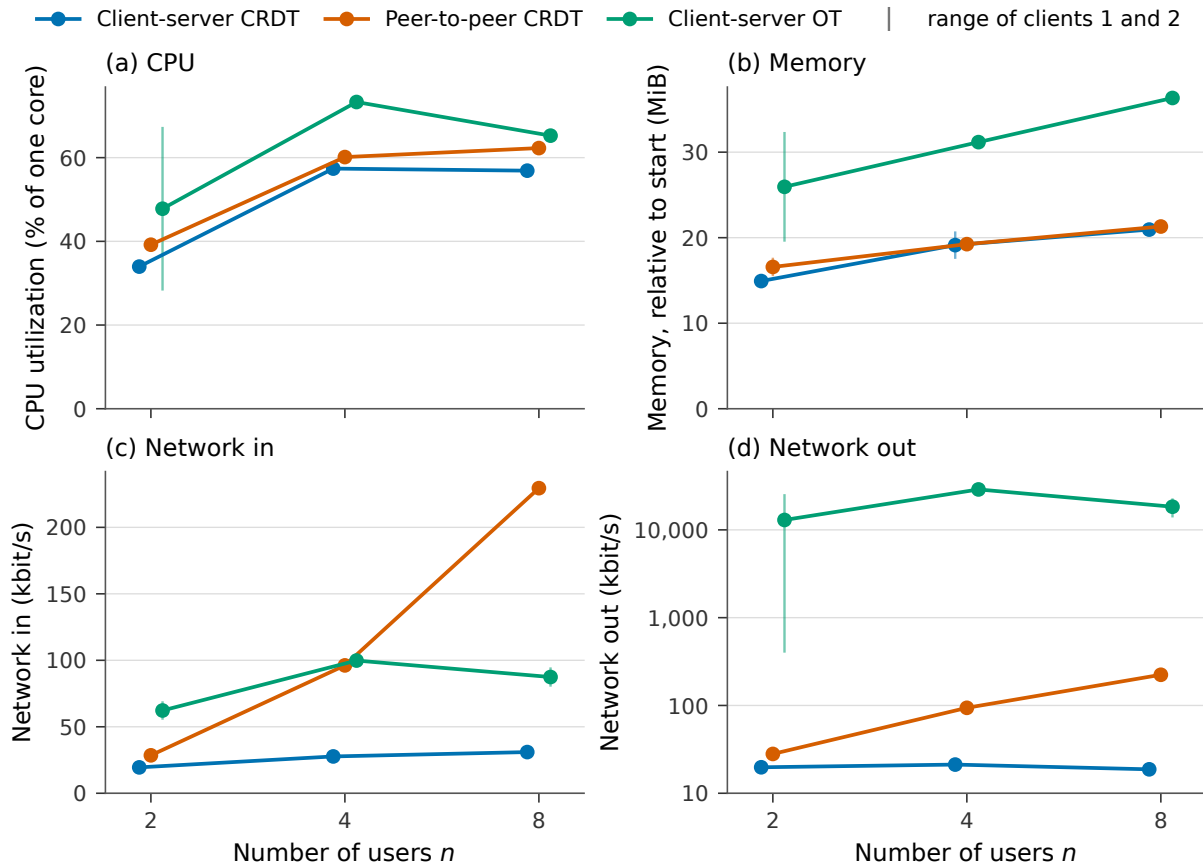


Figure 5.10. Mean resource usage of the client application in Scenario 1 against the number of users n (Table 5.3). Each point is the mean of the first two clients; the thin bars span the two clients' own values. Network out is on a logarithmic scale.

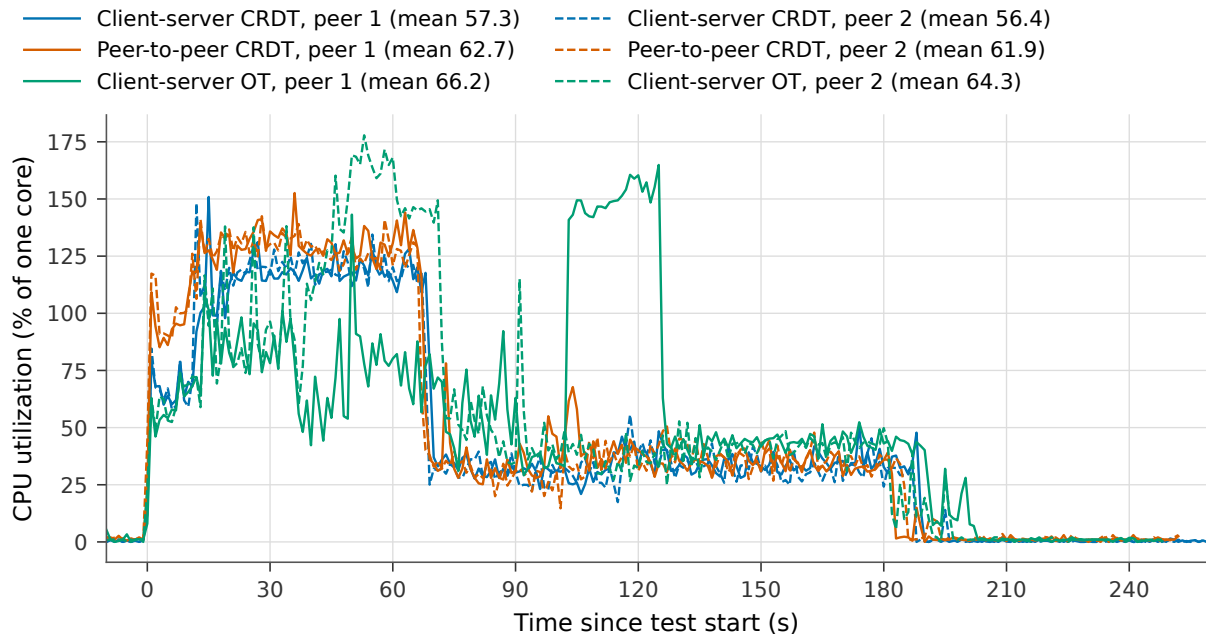


Figure 5.11. CPU utilization of the first and second clients, Scenario 1, $n = 8$.

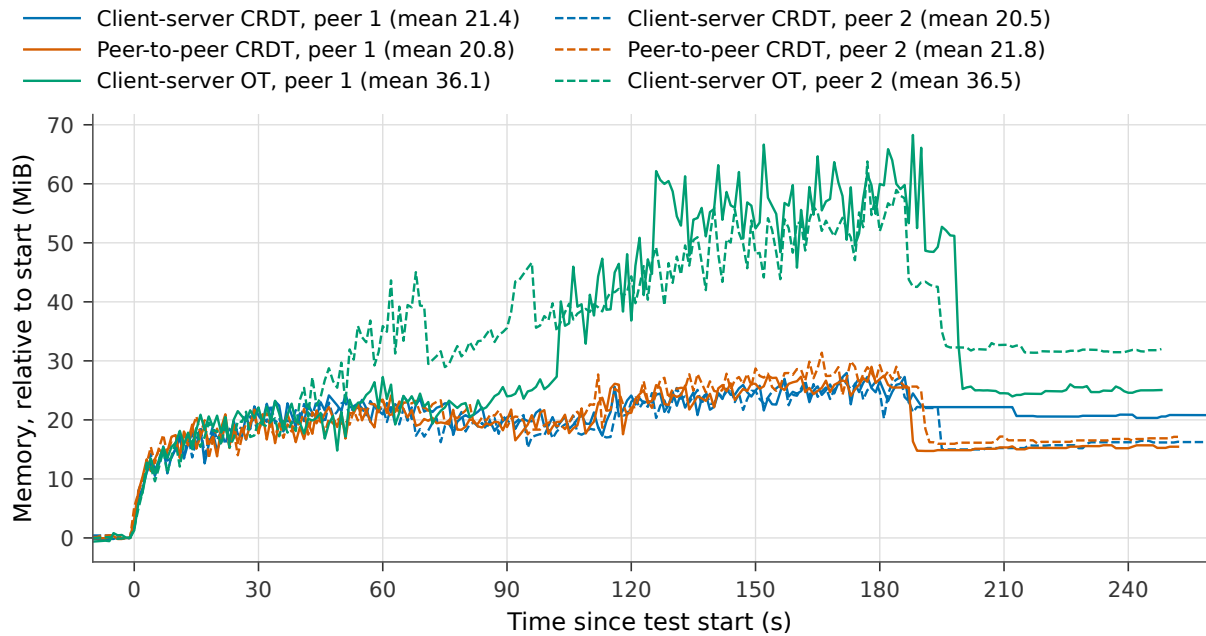


Figure 5.12. Memory growth of the first and second clients, Scenario 1, $n = 8$.

machine to the point of disrupting the user's work, every variant of PeerToCP meets the lightweight aspect in terms of CPU and RAM.

Table 5.4. Mean resource usage of the servers in Scenario 1, in the same units as Table 5.3. For the peer-to-peer variant, the server is the signaling server.

Variant	Metric	$n = 2$	$n = 4$	$n = 8$
Peer-to-peer CRDT	CPU	0.01	0.02	0.10
	RAM	0.67	−0.03	0.40
	Net in	10.72	14.81	60.72
	Net out	10.24	15.79	83.11
Client-server CRDT	CPU	0.93	1.29	1.68
	RAM	5.01	6.83	9.54
	Net in	73.26	176.64	324.19
	Net out	71.79	193.90	391.66
Client-server OT	CPU	4.85	14.18	24.33
	RAM	30.92	41.88	69.99
	Net in	52,318.08	167,768.04	308,540.87
	Net out	26,494.76	84,932.31	156,178.28

Resource usage on the server is the other half of the lightweight aspect (Table 5.4). The WebRTC signaling server of the peer-to-peer CRDT variant used minimal RAM and CPU. In the client-server architectures the server does much more: besides maintaining the connections of the clients in a network group, it keeps their replicas consistent. The server of the client-server CRDT variant sends and receives 1.4 to 2.6 times as much data as a single peer in the peer-to-peer variant.

The Yjs CRDT keeps a logical timestamp in the form of a state vector, which records the version of every user in the network. When two users synchronize, only the difference between their versions, derived from their state vectors, is sent; if the documents differ too much, the whole document may be sent instead, which can be smaller. Changes derived from the state vector are commutative and idempotent: the same change can be applied repeatedly and in any order without affecting the convergence of the document. Because of these properties, the data sent to each client does not depend on the number of operations performed, but on the size of the changes at the time of synchronization.

In Figure 5.14, if client A first synchronizes with client B, then client A synchronizes with client C, and then client B synchronizes with client C, client C only needs to send B what B has not already received from client A. And if an object in the data structure is deleted before it has been synchronized, its content never needs to be sent at all. These are among the Yjs optimizations that reduce the total bandwidth between users in the CRDT variants.

Some reference implementations add an external database to store documents in the client-server architecture, so that a network group can hold more documents. None of the variants here uses a third-party database; each keeps only simple built-in data structures on the server. In the peer-to-peer architecture, documents can instead be persisted in the local storage of each peer.

The price of the client-server CRDT variant's light traffic on the clients is heavier traffic on the server. In the peer-to-peer variant, on the other hand, the number of data channel connections grows quadratically with the number of users (Figure 5.15), so the total traffic each peer must handle grows faster with the

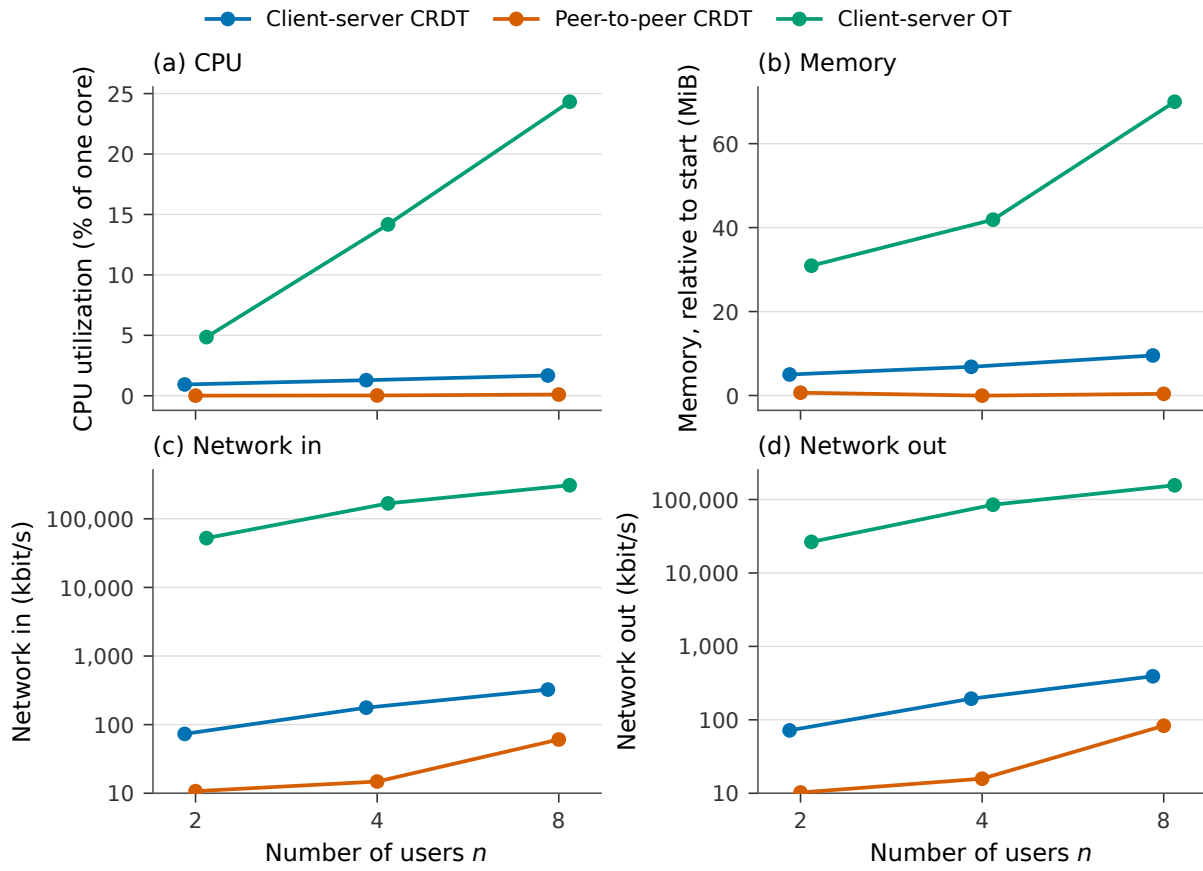


Figure 5.13. Mean resource usage of the servers in Scenario 1 against the number of users n (Table 5.4). For the peer-to-peer variant, the server is the signaling server. Network in and out are on logarithmic scales.

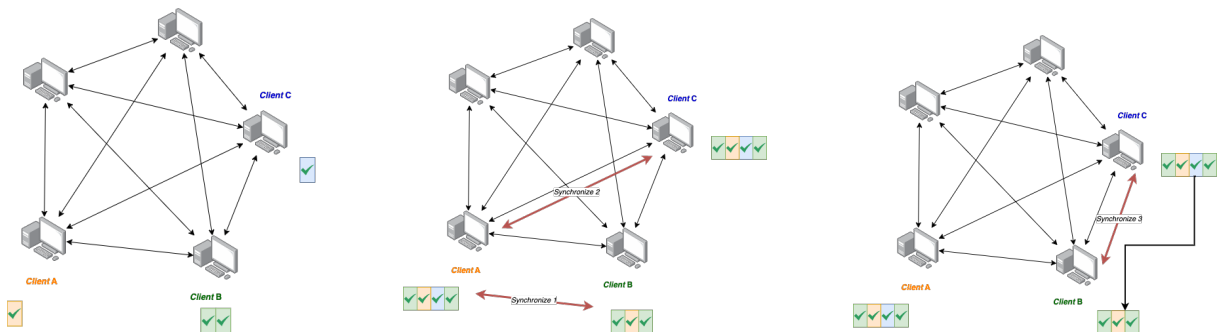


Figure 5.14. Synchronization in the peer-to-peer CRDT variant. Client A synchronizes with client B, then with client C; when B and C then synchronize, C only sends what B has not already received from A.

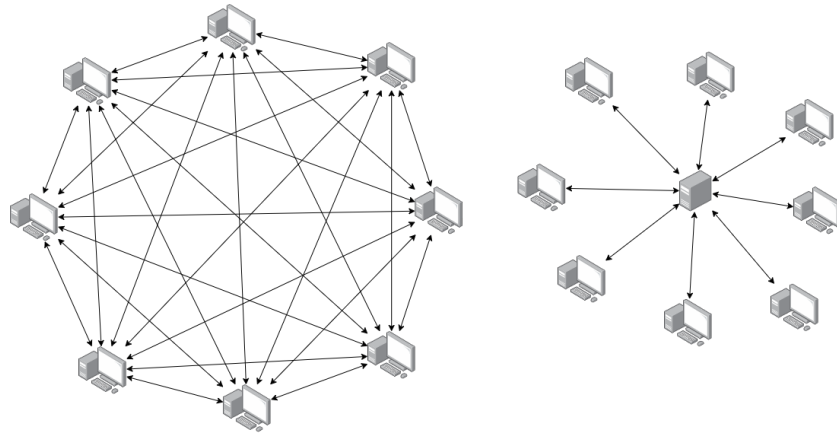


Figure 5.15. Connections in a peer-to-peer full mesh (left) and in a client-server star (right), for eight users.

size of the network group. The OT variant, given its conflict-resolution protocol, has by far the largest network load of all the variants, which also drives up its CPU and RAM usage. From the lightweight perspective, the client-server OT variant is not preferred, and its data transmission protocol needs to be revisited and optimized. More sophisticated and better-optimized OT algorithms also deserve further study, so that intensive editing such as in Scenario 1 can be handled.

Conclusion

This chapter concludes the study. It summarizes how each variant of PeerToCP was implemented and what the evaluation implies, and it outlines further development and experiments that could address the weaknesses found and lead to better systems of this kind.

6.1 Conclusions

This study set out to build PeerToCP, a real-time collaborative code editor with a shared shell. The application keeps data consistent in a distributed system with several architectures and algorithms: the operational transformation (OT) algorithm on a client-server architecture, the conflict-free replicated data type (CRDT) data structure on a client-server architecture, and a CRDT on a WebRTC-based peer-to-peer architecture.

The OT variant of PeerToCP relies on transformation property 1 (TP1) to keep every user's data in a network consistent. Every change is represented as an update operation, and both clients and server store the sequence of these operations. A client synchronizes its document with the other clients in the network by pulling updates from the server. A client may push updates to the server only if it already has the server's latest version. Otherwise, the client receives the server's newest operations and applies operational transformation to all of its local operations that the server has not yet accepted. The shared shell follows the same approach as the code editor, but its operational transformation is taken to run in constant time, because every operation is an append to an array.

Besides OT, there is the CRDT variant, which is essentially a further development of the OT algorithm with an additional data structure. PeerToCP uses the Yjs library, which implements the YATA CRDT. Every character in this data structure has its own ID. YATA stores the characters as a linear sequence, in which the position of a character is represented by its relation to the characters before and after it. This data structure is connected to a client-server network provider through the WebSocket-based y-websocket library. The peer-to-peer CRDT variant instead uses the WebRTC-based y-webrtc library, which uses WebSocket to talk to its signaling server. Both network providers give Yjs an abstraction for communicating with the other users in the network and synchronizing documents.

Every variant of PeerToCP has been shown correct theoretically, by the developers of the code it builds on, and empirically, through the four scenarios of this study. The main variant, the peer-to-peer CRDT, achieved the best latency and lower resource usage in the user's application. The OT variant suffered from excessive network bandwidth, because of a suboptimal protocol and poor concurrency of updates. This happens when every client in a network group keeps pushing updates at the same time. The server then tends to reject the update of a client whose local operations have piled up, because it gives priority to the version increments of other clients that push smaller updates.

Each variant also has its own strengths and weaknesses in how lightweight it is, on the server and in the user's application. Based on the test scenarios of this experiment, for groups of $n \leq 8$ users located close to one another, PeerToCP with the WebRTC-based peer-to-peer CRDT is the best choice. Its server load is also much lower than in the client-server architectures: in the peer-to-peer variant every computation happens on the users' own machines, so the signaling server only handles users connecting to and disconnecting from a network.

For larger network groups, the client-server CRDT variant is worth considering, because its data transmission grows more slowly with the number of clients. It is also an option when some users in the network cannot keep a WebRTC connection, because they are far apart, or because a network mechanism prevents them from connecting directly to other peers.

6.2 Future Work

The results suggest developing the system further into an adaptive network that changes with its conditions. This could make the service more reliable for all its users. On the algorithm side, other, more efficient variants of operational transformation or CRDTs could replace the ones PeerToCP uses now, to lower latency and the resources the application needs. One promising optimization of the CRDT variants is to modify the CRDT map (dictionary) that stores the shell replicas: it should exploit the fact that data is only ever appended to the end of an array, with no deletions or changes at any other index.

Beyond algorithms, networking and web technologies also deserve attention. HTTP/3, the newest version of HTTP at the time of this study, provides WebTransport, which could be studied as a replacement for WebSocket. WebTransport aims to offer a better programming interface, with all the features of WebSocket and a lower latency. Scalability to larger numbers of users could be tested by changing the test harness so that it no longer drives the user interface end to end. The front end and the human-computer interaction (HCI) aspects of PeerToCP could also be improved and studied.

The application could also be extended to a web version that runs without a desktop framework such as Electron, with the drawback that a web client cannot host a shell for every user in the network to share. Other technologies and programming languages with better performance and lighter resource usage than Node.js and Chromium in Electron could replace the current framework; Tauri and Qt are among the alternatives considered for PeerToCP.

References

- Adeyeye, M., Makitla, I., & Fogwill, T. (2013). Determining the signalling overhead of two common webrtc methods: Jsn via xmlhttprequest and sip over websocket. In *2013 africon* (pp. 1–5).
- Aho, A. V., Sethi, R., & Ullman, J. D. (1985). *Compilers: Principles, techniques, and tools*. Reading, MA: Addison-Wesley.
- Alvestrand, H. T. (2021a, January). *Transports for WebRTC* (No. 8835). RFC 8835. RFC Editor. Retrieved from <https://www.rfc-editor.org/info/rfc8835> doi: 10.17487/RFC8835
- Alvestrand, H. T. (2021b, January). *WebRTC MediaStream Identification in the Session Description Protocol* (No. 8830). RFC 8830. RFC Editor. Retrieved from <https://www.rfc-editor.org/info/rfc8830> doi: 10.17487/RFC8830
- Arefin, S. S., Azad, I., & Kabir, H. (2013). Modified sack-tcp and some application level techniques to support real-time application. *International Journal of Electrical and Computer Engineering (IJECE)*, 6, 105–114.
- Attiya, H., Burckhardt, S., Gotsman, A., Morrison, A., Yang, H., & Zawirski, M. (2016). Specification and complexity of collaborative text editing. In *Proceedings of the 2016 acm symposium on principles of distributed computing* (pp. 259–268).
- Belomestnykh, O. (2010). *A rebuilt, more real time Google documents*. Google Developers Blog. Retrieved September 15, 2022, from <https://drive.googleblog.com/2010/04/a-rebuilt-more-real-time-google.html>
- Belshe, M., Peon, R., & Thomson, M. (2015). *Hypertext transfer protocol version 2 (HTTP/2)* (RFC No. 7540). Internet Engineering Task Force.
- Bishop, M. (2022, June). *HTTP/3* (No. 9114). RFC 9114. RFC Editor. Retrieved from <https://www.rfc-editor.org/info/rfc9114> doi: 10.17487/RFC9114
- Day-Richter, J. (2010). *What’s different about the new google docs: Making collaboration fast*. Google Developers Blog. Retrieved September 15, 2022, from <https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html>
- Dutton, S., et al. (2012). Getting started with webrtc. *HTML5 Rocks*, 23.
- Ellis, C. A., & Gibbs, S. J. (1989). Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on management of data - SIGMOD '89*. ACM Press. Retrieved from <https://doi.org/10.1145/67544.66963> doi: 10.1145/67544.66963
- Fette, I., & Melnikov, A. (2011). *The WebSocket protocol* (RFC No. 6455). Internet Engineering Task Force.
- Fielding, R., & Reschke, J. (2015). *Hypertext transfer protocol (http/1.1): Message syntax and routing, ietf rfc 7230*.
- Fietze, M. (2017). *Http/2 streams: Is the future of websockets decided?* Faculty of Computer Science, TU Dresden. <https://www.rn.inf.tu-dresden...>
- Firefox. (2022). *Spidermonkey documentation*. Firefox. Retrieved from <https://firefox-source-docs.mozilla.org/js/index.html>
- Frindell, A., Kinnear, E., & Vasiliev, V. (2022, July 6). *WebTransport over HTTP/3* (Internet-Draft No. draft-ietf-webtrans-http3-03). Internet Engineering Task Force. Retrieved from <https://datatracker.ietf.org/doc/draft-ietf-webtrans-http3/03/> (Work in Progress)
- Ganaputra, J., & Pardamean, B. (2015). Asynchronous publish/subscribe architecture over websocket for building real-time web applications. *Internetworking Indonesia*, 7(2), 15–19.
- Gentle, J. (2011). *ShareJS – Live concurrent editing in your app*. ShareJS. Retrieved September 16, 2022, from <https://sharejs.org/>
- Google. (2022). *V8 documentation*. Google. Retrieved from <https://v8.dev/docs>

- Harris, J. (2010). *What's different about the new Google Docs?* Google Developers Blog. Retrieved September 15, 2022, from <https://drive.googleblog.com/2010/05/whats-different-about-new-google-docs.html>
- IntelliJ, I. (2011). the most intelligent java ide. *JetBrains [online]*. [cit. 2016-02-23]. Dostupné z: <https://www.jetbrains.com/idea/#chooseYourEdition>.
- Jennings, C., Hardie, T., & Westerlund, M. (2013). Real-time communications for the web. *IEEE Communications Magazine*, 51(4), 20–26.
- Jesup, R., Loreto, S., & Tüxen, M. (2021, January). *WebRTC Data Channels* (No. 8831). RFC 8831. RFC Editor. Retrieved from <https://www.rfc-editor.org/info/rfc8831> doi: 10.17487/RFC8831
- Kinder, K. (2013). Sublime text: one editor to rule them all? *Linux Journal*, 2013(232), 2.
- Kleppmann, M., Gomes, V. B., Mulligan, D. P., & Beresford, A. R. (2019). Interleaving anomalies in collaborative text editors. In *Proceedings of the 6th workshop on principles and practice of consistency for distributed data* (pp. 1–7).
- Kredpattanakul, K., & Limpiyakorn, Y. (2018). Transforming javascript-based web application to cross-platform desktop with electron. In *International conference on information science and applications* (pp. 571–579).
- Krishnamurthy, B., Mogul, J. C., & Kristol, D. M. (1999). Key differences between http/1.0 and http/1.1. *Computer Networks*, 31(11-16), 1737–1751.
- Leibnitz, K., Hoßfeld, T., Wakamiya, N., & Murata, M. (2007). Peer-to-peer vs. client/server: Reliability and efficiency of a content distribution service. In *International teletraffic congress* (pp. 1161–1172).
- Li, D., & Li, R. (2004). Preserving operation effects relation in group editors. In *Proceedings of the 2004 ACM conference on computer supported cooperative work - CSCW '04*. ACM Press. Retrieved from <https://doi.org/10.1145/1031607.1031683> doi: 10.1145/1031607.1031683
- Lv, X., Cui, L., & Li, J. (2015). The research and design of real-time collaborative document management system. In *Proceedings of the 2015 3rd international conference on machinery, materials and information technology applications*. Atlantis Press. Retrieved from <https://doi.org/10.2991/icmmita-15.2015.160> doi: 10.2991/icmmita-15.2015.160
- Maly, R. J., Mischke, J., Kurtansky, P., & Stiller, B. (2003). Comparison of centralized (client-server) and decentralized (peer-to-peer) networking. *Semester thesis, ETH Zurich, Zurich, Switzerland*, 1–12.
- Martin, J. L. (2020). *Conflict-free replicated data types (CRDT) for distributed JavaScript apps*. TL;DR. Retrieved September 18, 2022, from <https://www.youtube.com/watch?v=M8-WFTjZoA0>
- Matthews, P., Rosenberg, J., & Mahy, R. (2010, April). *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)* (No. 5766). RFC 5766. RFC Editor. Retrieved from <https://www.rfc-editor.org/info/rfc5766> doi: 10.17487/RFC5766
- Miglani, S., & Shriram, S. (n.d.). *Electron: Build cross-platform desktop apps with JavaScript, HTML, and CSS*.
- Molli, P., Urso, P., Imine, A., et al. (2006). Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *2006 international conference on collaborative computing: Networking, applications and worksharing* (pp. 1–10).
- Nédelec, B., Molli, P., Mostefaoui, A., & Desmontils, E. (2013). Lseq: an adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 acm symposium on document engineering* (pp. 37–46).
- Nicolaescu, P., Jahns, K., Derntl, M., & Klamma, R. (2016, November). Near real-time peer-to-peer shared editing on extensible data types. In *Proceedings of the 19th international conference on supporting group work*. ACM. Retrieved from <https://doi.org/10.1145/2957276.2957310> doi: 10.1145/2957276.2957310
- OpenJS Foundation. (2022). *What is electron?* OpenJS Foundation and Electron contributors. Retrieved November 19, 2022, from <https://www.electronjs.org/docs/latest>
- Oster, G., Urso, P., Molli, P., & Imine, A. (2005). *Real time group editors without operational transformation* (Research Report No. RR-5580). INRIA.

- Perkins, C., Westerlund, M., & Ott, J. (2021, January). *Media Transport and Use of RTP in WebRTC* (No. 8834). RFC 8834. RFC Editor. Retrieved from <https://www.rfc-editor.org/info/rfc8834> doi: 10.17487/RFC8834
- Petit-Huguenin, M., Nandakumar, S., Holmberg, C., Keränen, A., & Shpount, R. (2021, January). *Session Description Protocol (SDP) Offer/Answer Procedures for Interactive Connectivity Establishment (ICE)* (No. 8839). RFC 8839. RFC Editor. Retrieved from <https://www.rfc-editor.org/info/rfc8839> doi: 10.17487/RFC8839
- Pimentel, V., & Nickerson, B. G. (2012). Communicating and displaying real-time data with websocket. *IEEE Internet Computing*, 16(4), 45–53.
- Preguiça, N. (2018). *Conflict-free replicated data types: An overview*. arXiv. Retrieved from <https://arxiv.org/abs/1806.10254> doi: 10.48550/ARXIV.1806.10254
- Preguiça, N., Baquero, C., & Shapiro, M. (2018). *Conflict-free Replicated Data Types (CRDTs)*. arXiv. Retrieved from <https://arxiv.org/abs/1805.06358> doi: 10.48550/ARXIV.1805.06358
- Ressel, M., Nitsche-Ruhland, D., & Gunzenhäuser, R. (1996). An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM conference on computer supported cooperative work - CSCW '96*. ACM Press. Retrieved from <https://doi.org/10.1145/240080.240305> doi: 10.1145/240080.240305
- Reynolds, F. (2008). Web 2.0—in your hand. *IEEE Pervasive Computing*, 8(1), 86–88.
- Roh, H.-G., Jeon, M., Kim, J.-S., & Lee, J. (2011). Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3), 354–368.
- Shapiro, M., Preguiça, N., Baquero, C., & Zawirski, M. (2011). Conflict-free replicated data types. In *Lecture notes in computer science* (pp. 386–400). Springer Berlin Heidelberg. Retrieved from https://doi.org/10.1007/978-3-642-24550-3_29 doi: 10.1007/978-3-642-24550-3_29
- Smith, Z. (2012). *Overview of operational transformation*. University of Minnesota. Retrieved September 18, 2022, from <https://umm-csci.github.io/senior-seminar/seminars/spring2012/Smith.pdf>
- Sredojević, B., Samardžija, D., & Posarac, D. (2015). WebRTC technology overview and signaling solution design and implementation. In *2015 38th international convention on information and communication technology, electronics and microelectronics (mipro)* (pp. 1006–1009).
- Srinivasan, R. (1995). *RPC: Remote procedure call protocol specification version 2* (RFC No. 1831). Internet Engineering Task Force.
- Stenberg, D. (2014). *Http2 explained* (Vol. 44) (No. 3). ACM New York, NY, USA.
- Sun, C., & Ellis, C. (1998). Operational transformation in real-time group editors. In *Proceedings of the 1998 ACM conference on computer supported cooperative work - CSCW '98*. ACM Press. Retrieved from <https://doi.org/10.1145/289444.289469> doi: 10.1145/289444.289469
- Sun, C., Sun, D., Agustina, & Cai, W. (2019). *Real differences between OT and CRDT under a general transformation framework for consistency maintenance in co-editors*. arXiv. Retrieved from <https://arxiv.org/abs/1905.01518> doi: 10.48550/ARXIV.1905.01518
- Sun, C., Xu, Y., & Ng, A. (2017, February). Exhaustive search and resolution of puzzles in OT systems supporting string-wise operations. In *Proceedings of the 2017 ACM conference on computer supported cooperative work and social computing*. ACM. Retrieved from <https://doi.org/10.1145/2998181.2998252> doi: 10.1145/2998181.2998252
- Sun, D., Sun, C., Ng, A., & Cai, W. (2019a). *Real differences between ot and crdt in correctness and complexity for consistency maintenance in co-editors*. arXiv. Retrieved from <https://arxiv.org/abs/1905.01302> doi: 10.48550/ARXIV.1905.01302
- Sun, D., Sun, C., Ng, A., & Cai, W. (2019b). *Real differences between OT and CRDT in building co-editing systems and real world applications*. arXiv. Retrieved from <https://arxiv.org/abs/1905.01517> doi: 10.48550/ARXIV.1905.01517
- Sun, D., Xia, S., Sun, C., & Chen, D. (2004). Operational transformation for collaborative word processing. In *Proceedings of the 2004 ACM conference on computer supported cooperative work - CSCW '04*. ACM Press. Retrieved from <https://doi.org/10.1145/1031607.1031681> doi: 10.1145/1031607.1031681

- Thompson, S. (2015). *Smarter garbage collection for smoother browsing and less memory usage*. Chromium Developers Blog. Retrieved November 12, 2022, from <https://blog.chromium.org/2015/12/smarter-garbage-collection-for-smoother.html>
- Tilkov, S., & Vinoski, S. (2010). Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6), 80–83.
- Vidot, N., Cart, M., Ferrié, J., & Suleiman, M. (2000). Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the 2000 ACM conference on computer supported cooperative work - CSCW '00*. ACM Press. Retrieved from <https://doi.org/10.1145/358916.358988> doi: 10.1145/358916.358988
- Weiss, S., Urso, P., & Molli, P. (2009). Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *2009 29th IEEE international conference on distributed computing systems* (pp. 404–412).
- Xu, Y., & Sun, C. (2016, March). Conditions and patterns for achieving convergence in OT-based co-editors. *IEEE Transactions on Parallel and Distributed Systems*, 27(3), 695–709. Retrieved from <https://doi.org/10.1109/tpds.2015.2412938> doi: 10.1109/tpds.2015.2412938

Code and Experimental Setup

All code, the client application and the test experiments of this study are available at <https://github.com/hockyy/peertocp>. Each variant of PeerToCP is a separate branch: `crdt-cs`, the client-server CRDT variant; `ot-cs`, the client-server operational transformation variant; and `crdt-p2p`, the peer-to-peer CRDT variant. Instructions for rerunning the tests and their scenarios are in the README.md of the `crdt-p2p` branch, the repository's default branch. The servers and the modified connection providers used by each branch are available at:

- `crdt-cs`: <https://github.com/hockyy/y-websocket>;
- `crdt-p2p`: <https://github.com/hockyy/y-webrtc>;
- `ot-cs`: <https://github.com/hockyy/peertocp-ot-server>.

For the experiments, every instance was set up with the following commands.

```
sudo apt install -y git wget screen nginx python-is-python3 g++ make
sudo apt install -y build-essential clang libdbus-1-dev libgtk2.0-dev \
    libnotify-dev libgnome-keyring-dev libgconf2-dev \
    libasound2-dev libcap-dev libcups2-dev libxtst-dev \
    libxss1 libnss3-dev gcc-multilib g++-multilib libasound2 xvfb
export DISPLAY=192.168.0.5:0.0
curl https://my-netdata.io/kickstart.sh > /tmp/netdata-kickstart.sh && sh /tmp/netdata-kickstart.sh
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.2/install.sh | bash
source ~/.bashrc
nvm install 16
nvm use 16
git clone [URL]

# Use xvfb, because Debian has no desktop
xvfb-run npm start
```

The measurements were downloaded and processed on a local machine with the following script, which fetches the evaluation logs.

```
netd () {
    curl "http://$1:19999/api/v1/data?chart=apps.mem&dimension=node \
        &after=$2&points=0&group=average&gtime=0 \
        &timeout=0&format=csv&options=seconds" \
        > mem-$3.csv
    curl "http://$1:19999/api/v1/data?chart=system.ip \
        &after=$2&points=0&group=average&gtime=0 \
        &timeout=0&format=csv&options=seconds" \
        > network-$3.csv
    curl "http://$1:19999/api/v1/data?chart=apps.cpu&dimension=node \
```

```
        &after=$2&points=0&group=average&time=0 \
        &timeout=0&format=csv&options=seconds" \
        > cpu-$3.csv
    }

    scpd () {
        scp -r hocky@$1:~/peertocpnext/out/ ./ $2
    }
```

Every server ran NGINX with the following configuration.

```
location / {
    # First attempt to serve request as file, then
    # as directory, then fall back to displaying a 404.
    # try_files $uri $uri/ =404;
    proxy_pass http://127.0.0.1:3000;

    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_set_header Host $http_host;
    proxy_set_header X-Real-IP $remote_addr;
}
```

The raw measurements, and the notebook that visualizes and interprets them, are available at <https://github.com/hockyy/peertocp-benchmark>.