

Joshua Send

# **Conflict Free Document Editing with Different Technologies**

Computer Science Tripos – Part II

Trinity Hall

17th May 2017



# Proforma

Name: **Joshua Send**  
College: **Trinity Hall**  
Project Title: **Conflict Free Document Editing with  
Different Technologies**  
Examination: **Computer Science Tripos – Part II, June 2017**  
Word Count: **11825**  
Project Originator: Joshua Send  
Supervisor: Stephan Kollmann

## Original Aims of the Project

To design, test, and analyze a collaborative text editor using convergent replicated data type (CRDT) in comparison to a text editor implemented with operational transformations (OT). The CRDT-based system communicates through a simulated network, while the OT-based system is built around the existing Javascript library ShareJS and utilizes a standard network stack. The relative efficiency of various aspects is compared by running a series of experiments on both systems.

## Work Completed

The main success criteria have been fulfilled. The system I built implements a thoroughly tested CRDT for text editing on top of a general network simulation delivering packets across an arbitrarily defined topology. In addition, I completed an optional extension adding undo and redo functionality to my CRDT, which has not been done before. The alternative system built around ShareJS was adapted to run the same experiments as my custom implementation. The results show the strength of my CRDT in highly concurrent and replicated situations, while ShareJS has the advantage in some metrics for very large text documents.

## Special Difficulties

None.

## Declaration

I, Joshua Send of Trinity Hall, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date



# Contents

<b>1</b>	<b>Introduction</b>	<b>13</b>
1.1	Motivation . . . . .	13
1.2	Overview . . . . .	14
<b>2</b>	<b>Preparation</b>	<b>15</b>
2.1	Consistency Models . . . . .	15
2.1.1	Definition of “Conflict Free” . . . . .	15
2.1.2	CCI Consistency Model . . . . .	15
2.2	Achieving Eventual Consistency . . . . .	16
2.2.1	Operational Transformations . . . . .	16
2.2.2	ShareJS . . . . .	18
2.2.3	Convergent Replicated Data Types . . . . .	19
2.2.4	CRDTs for Text Editing . . . . .	20
2.3	Starting Point . . . . .	22
2.4	Requirements Analysis . . . . .	23
2.5	Software Engineering . . . . .	23
2.5.1	Libraries . . . . .	23
2.5.2	Languages and Tooling . . . . .	24
2.5.3	Backup Strategy and Development Machine . . . . .	24
2.6	Early Design Decisions . . . . .	24
2.6.1	Network Simulation . . . . .	24
2.6.2	Data Collection and Logging . . . . .	25
<b>3</b>	<b>Implementation</b>	<b>27</b>
3.1	CRDT-based system . . . . .	27
3.1.1	Overview . . . . .	27
3.1.2	CRDT . . . . .	28
3.1.3	Simulated Network . . . . .	33
3.1.4	Simulation . . . . .	37
3.2	ShareJS Comparative Environment . . . . .	39
3.2.1	Scheduling . . . . .	39
3.2.2	Logging . . . . .	39
3.3	Experiment Creation and Use . . . . .	39
3.3.1	Work Flow . . . . .	39

3.3.2	Experiment Design . . . . .	40
3.3.3	Separation of Modeling and Analysis . . . . .	41
3.4	Extension: Local Undo . . . . .	41
3.4.1	Overview . . . . .	41
3.4.2	Undo and Redo of Insertions . . . . .	41
3.4.3	Undo and Redo of Deletions . . . . .	42
3.5	Summary of Implementation . . . . .	45
<b>4</b>	<b>Evaluation</b>	<b>47</b>
4.1	Overall Results . . . . .	47
4.2	Testing the CRDT . . . . .	48
4.3	Quantitative Analysis . . . . .	48
4.3.1	Memory . . . . .	49
4.3.2	Network . . . . .	54
4.3.3	CPU Time . . . . .	59
<b>5</b>	<b>Conclusion</b>	<b>61</b>
5.1	Achievements . . . . .	61
5.2	Lessons Learned . . . . .	62
5.3	Limitations and Future Work . . . . .	62
	<b>Bibliography</b>	<b>63</b>
<b>A</b>	<b>Vector Clocks</b>	<b>69</b>
A.1	Formal Definition . . . . .	69
A.2	Modified Vector Clock . . . . .	69
<b>B</b>	<b>Convergence of Immediate Undo Variant</b>	<b>71</b>
B.1	Proof of Commutativity . . . . .	71
<b>C</b>	<b>Simple Experiment Experimental Setup and Results</b>	<b>73</b>
C.1	Experiment Setup . . . . .	73
C.2	Summary of Results of Simple Experiment . . . . .	74
<b>D</b>	<b>Project Proposal</b>	<b>77</b>



# List of Figures

2.1	Achieving Conflict Free Text Editing . . . . .	16
2.2	Operational Transformations — concurrent insertion . . . . .	17
2.3	Operational Transformations — concurrent deletion . . . . .	17
2.4	Operational Transformations — concurrent insertion and deletion . . . . .	18
2.5	Text CRDT as a tagged set . . . . .	20
2.6	Concurrent Updates in Treedoc . . . . .	21
3.1	System Architecture . . . . .	28
3.2	Updating linked lists . . . . .	29
3.3	Annotated CRDT . . . . .	29
3.4	Convergence using Total Order . . . . .	31
3.5	Ensuring Causal Delivery . . . . .	36
3.6	Workflow . . . . .	40
4.1	Unit tests for CRDT . . . . .	48
4.2	Memory Consumption Sanity Check . . . . .	50
4.3	Single Client Memory Consumption . . . . .	51
4.4	Memory Consumption versus Replication . . . . .	52
4.5	Behavior of System Variants - Memory . . . . .	53
4.6	Client-Server Latency . . . . .	54
4.7	Server-Client Latencies . . . . .	55
4.8	Behavior of System Variants - Packet Size . . . . .	59
4.9	CPU Time . . . . .	60



# List of Tables

2.1	Project (Sub)goals, priority, and difficulty . . . . .	23
4.1	ShareJS Insert and Delete Packet Size . . . . .	55
4.2	CRDT System Insert Packet Size . . . . .	56
4.3	Sample Packet Sizes . . . . .	57
4.4	Number of Packets in P2P Network . . . . .	57

## Acknowledgements

I would like to thank the following people for their helpful and enthusiastic contributions to this project:

- **Stephan Kollmann**, for continual support and guidance throughout the project and for often hinting at ideas I may have missed and thereby letting me explore and develop my own solutions.
- **Martin Kleppmann**, for his willingness to help with various aspects of the project, and especially providing the starting point for development of the CRDT.
- **Joseph Gentle**, for providing the initial idea of implementing a P2P distributed text editor. Though the idea changed in terms of technology, the core idea remained.
- **Proof Readers** of this document including Prof. Simon Moore, my family and my friends.

# Chapter 1

## Introduction

Real time interaction between users is becoming an increasingly important feature in many applications, from word processing to CAD to social networking. This dissertation examines trade-offs that should be considered when applying the prevailing technologies that enable lock free, distributed modification of data. More specifically, this project implements and analyzes a concurrent text editor based on Convergent Replicated Data Types (also known as Conflict-free Replicated Data Types), CRDT in short, in comparison to an existing editor exploiting Operational Transformations (OT) as its core technology.

### 1.1 Motivation

Realtime collaborative editing was first demonstrated in the Mother of All Demos by Douglas Engelbart in 1968 [31]. From that time, it took several decades for commercial implementations of such editing systems to appear. Early products were released in the 1990s and the 1989 paper by Gibbs and Ellis [8] marked the beginning of extended research into operational transformations (OT). After almost 20 years of research, OT is a relatively developed field and can be found in commonly available products, though many early algorithms have been shown to have flaws [21].

The most familiar OT-based product is probably Google Docs<sup>1</sup>, which is generally agreed upon to behave in a predictable and well understood way. Importantly, it follows users' expectations for how a concurrent, multi-user document editor should work. This includes lock-free editing, no loss of data, and the guarantee that everyone converges to the same document once modifications have been exchanged. However, the OT algorithms powering distributed text editing are error-prone and require high amounts of processing power, as shown in [4]. Conflict-free replicated data types (CRDTs) offer similar features with simpler theoretical foundations and better performance.

Both CRDTs and OT provide eventual consistency: the guarantee that shared data converges to the same state eventually, as in Google Docs. This relates to a cornerstone of

---

<sup>1</sup><https://docs.google.com>

distributed systems, the CAP theorem: one cannot guarantee all three of consistency, availability, and partition-tolerance [13]. However, relaxing the consistency constraint to eventual consistency enables all three [36]. Providing eventual consistency is non-trivial and achieved by both OT and CRDTs at different costs, as explored by this dissertation.

## 1.2 Overview

This project aims to examine the compromises made when implementing highly distributed and concurrent document editing with Convergent Replicated Data Types (CRDTs) versus with Operational Transformations (OT). To do this, I have designed experiments which expose statistics about network and processor usage, memory consumption, and scalability. These experiments are run on a system I created based on a specific CRDT, and on a comparative environment built around the open source library ShareJS that implements OT. The system meets the originally proposed goals of implementing a concurrent text editor based on CRDTs which passes various tests for correctness; quantitative analysis is presented in the Evaluation section.

In contrast to the OT-based library ShareJS, my system, detailed in §3.1, runs on a peer to peer network architecture instead of a traditional client-server model. The lack of a server reduces the number of stateful parts in the system, at the expense of more complex networking. I managed this complexity by using a simulated peer to peer network, allowing me to control the precise topology, link latencies, and protocol, and to explore advantages and disadvantages of using P2P.

In addition to implementing, testing, and analyzing an editor based on a CRDT, as an extension I also upgraded its functionality to include undo and redo capabilities. I developed two approaches using different semantics and consistency models originally. However, one paper I discovered, which develops Logoot-Undo [33], takes a very similar approach and is discussed below.

Part of the challenge of this project was to develop the core CRDT and associated algorithms based only on a high level explanation of the desired functionality provided by Martin Kleppmann, based on a paper by Roh *et al.* [25] (described in the next chapter). My solution mirrors and incidentally simplifies some of the work done in the paper, and leads to several interesting possibilities for future work (§5.3).

# Chapter 2

## Preparation

### 2.1 Consistency Models

#### 2.1.1 Definition of “Conflict Free”

There appear to be multiple interpretations of “conflict free”. On one hand, there is the users’ intuitive idea that any of their own operations should behave as if they were the only users on the system. For example, inserting a character at index 9 should result in it appearing and staying at index 9. On the other hand, there is the data-centric view of conflict. In this case, operations conflict if they are concurrent and modify the same data. Two users inserting different characters at index 10 concurrently have made conflicting operations. Conflict free then means that no data is lost, and after all users’ operations are exchanged the resulting text agrees.

To demonstrate, in the situation above each user expects their text to appear at index 9. If this were carried out, one writer would win in a classic last-writer-wins scenario. However, to satisfy the data-centric definition, data cannot be lost. The solution is to let one user ‘win’ and insert their characters index 9, and offset the other user’s characters to appear later. Both OTs and CRDTs achieve this merge in fundamentally different ways. This entire process is demonstrated in Figure 2.1.

#### 2.1.2 CCI Consistency Model

The commonly used consistency model for concurrent document editing is the CCI model. The definition here is borrowed from [33].

- **Consistency:** All operations ordered by a precedence relation, such as Lamport’s happened-before relation [17], are executed in the same order on every replica.
- **Convergence:** The system converges if all replicas are identical when the system is idle.

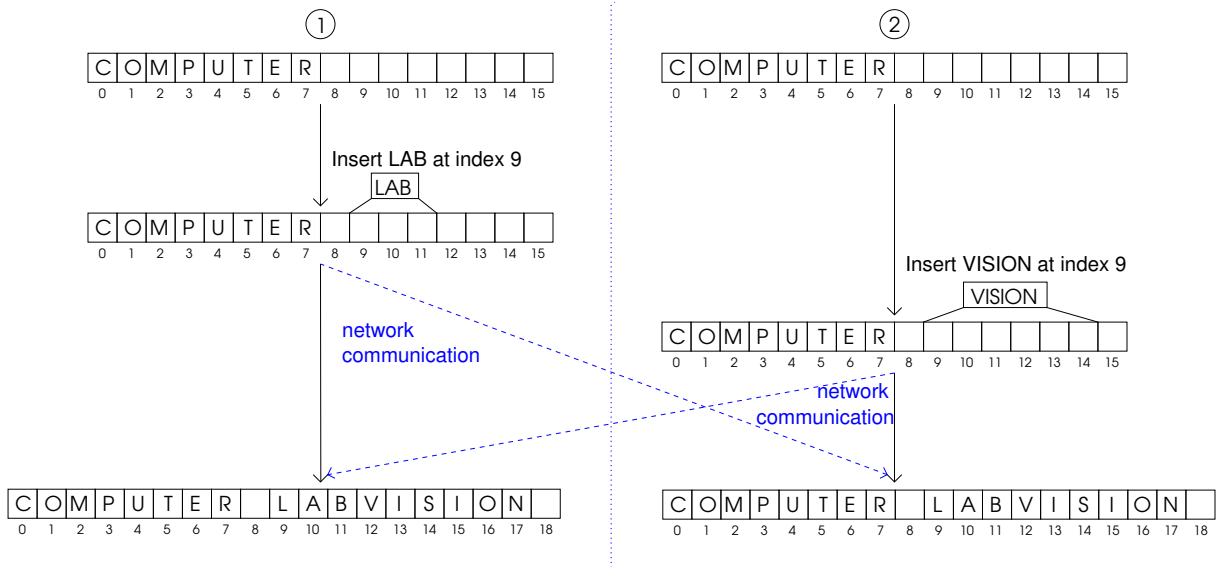


Figure 2.1: Two clients are initially in a quiescent state i.e. the system has settled with the shared string “computer”. They then concurrently insert different words at the same index. At first each sees only their own edit. Then operations are exchanged and the system reaches quiescence again. Both clients see the string “computer labvision”. Client 1 thus ‘won’ and kept the original index, while Client 2 had its insertion offset.

- **Intention Preservation:** The expected effect of an operation should be observed on all replicas. This is commonly accepted to mean:
  - *delete* A deleted line must not appear in the document unless the deletion is undone.
  - *insert* A line inserted on a peer must appear on every peer and the order relation between the document lines and a newly inserted line must be preserved.
  - *undo* Undoing a modification makes the system return to the state it would have reached if this modification was never produced.

The given definition of intention preservation is widely used in CRDT development, but may produce some unexpected results as seen when implementing undo functionality in §3.4.3.

## 2.2 Achieving Eventual Consistency

### 2.2.1 Operational Transformations

The easiest way to understand operational transformations is with examples of commonly encountered conflicting operations.

The following figure demonstrates the *concurrent insert at the same index* case, in which two clients at different sites concurrently insert text into the same index in the shared



buffer. The key element is the *transform* function which recognizes a local insertion at the same places as the remote one and chooses to offset or not offset it based on client IDs.

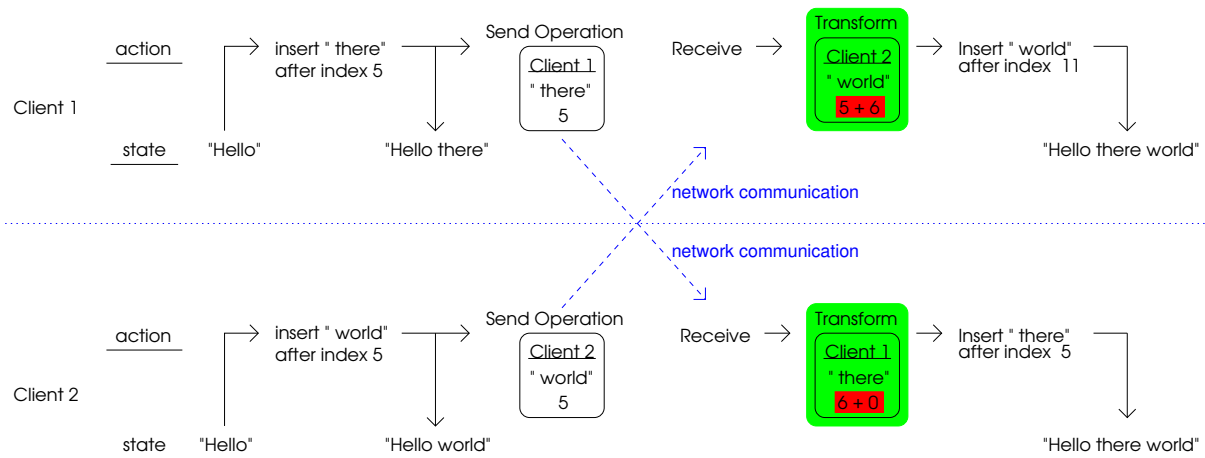


Figure 2.2: This figure shows how operational transformations might handle concurrent insertion at the same index. Both clients insert different strings after index 5. The operations are exchanged, and the key *transform* function (in green) detects the conflict, and chooses to offset “ world” on Client 1 by 6 (in red), which is the length of its own previously inserted string “ there”. On Client 2, once the insert “ there” arrives, the algorithm knows not to offset it (due to some arbitrary ordering such as client ID) and places it at index 5. Thus both clients resolve the string “Hello there world”

The next figure demonstrates the *concurrent deletion of the same character* case.

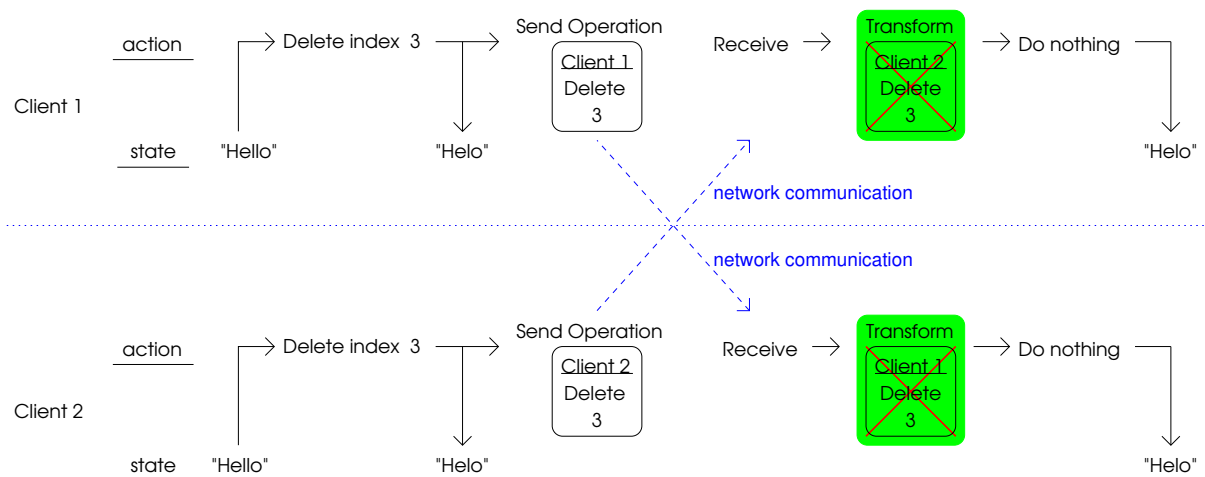


Figure 2.3: Both clients concurrently delete index 3 of “Hello”, resulting in “Helo”. The operations are exchanged. The *transform* function (in green) detects the conflict, and on both clients discards the remote operation. Integrating it would cause modifications the user did not execute (i.e. delete ‘o’ in addition). By discarding the operations, both clients resolve “Helo” correctly.

This final figure demonstrates the *concurrent insertion and deletion* case.

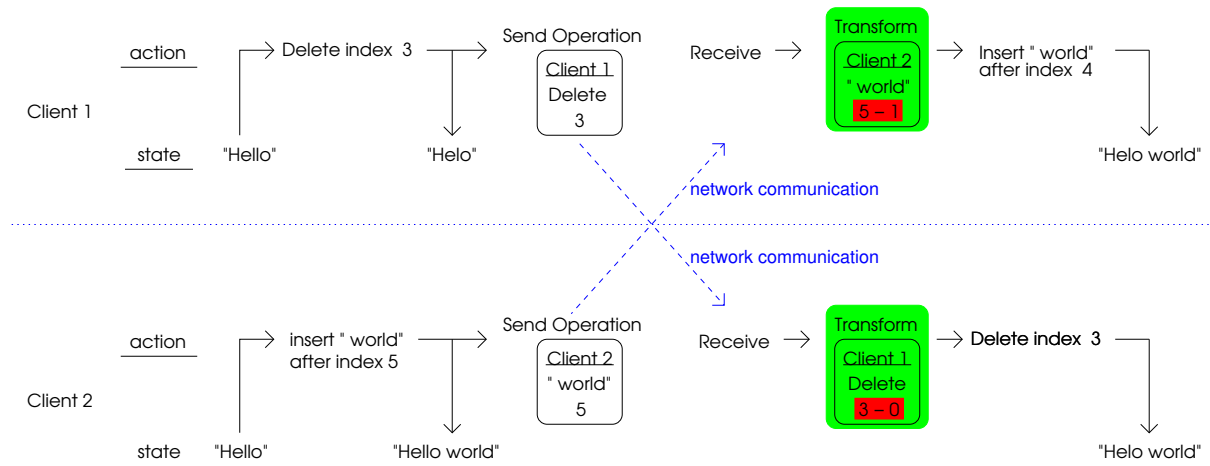


Figure 2.4: Client 1 deletes index 3 of the initial string “Hello”, resulting in “Helo”. Meanwhile, Client 2 inserts “ world” after index 5 resulting in “Hello world”. The operations are exchanged. The *transform* function (in green) detects the conflict, and on Client 1 knows to subtract 1 from the index to insert “ world” after because a prior character has been deleted concurrently. On Client 2, there is no conflict and the delete can proceed at position 3. Thus both clients resolve “Helo World” correctly.

## 2.2.2 ShareJS

ShareJS [12] is an open source Javascript library implementing Operational Transformations which can be deployed on web browsers or NodeJS<sup>1</sup> clients. As there are a large variety of algorithms that can enable OT [16], rather than tracking down the papers ShareJS is based on, much of what is summarized below was deduced by reading its source code. Its core features are shared, versioned documents, an active server which orders and transforms operations, and primary actions *insert* and *delete*.

Each operation applies to a specific document version. The version number is used to detect concurrent changes and transform operations against each other.

The following shows an *insert* operation for inserting “Hello World” at index 100 in document version 1, transmitted as JSON<sup>2</sup>:

```
{v:1, op:[{i:'Hello World', p:100}]}
```

A deletion of the word “Hello” at index 100:

```
{v:1, op:[{d:'Hello', p:100}]}
```

Multiple operations may be sent in one packet:

```
{v:1, op:[{d:'World', p:100}, {i:'Cambridge', p:110}]}
```

<sup>1</sup><https://nodejs.org/en/>

<sup>2</sup>JavaScript Object Notation – a human readable and writable serialization of Javascript objects using maps and lists (<http://www.json.org/>)

The library contains both client and a server code. The server provides a serialized order of operations to be applied on each client. The server also transforms concurrent operations against each other, but has the choice of rejecting an operation if the target document version is too old. In order to transform operations against each other, the server must maintain a list of past operations, which is likely to be the primary source of memory consumption. This is confirmed in §4.3.1.

ShareJS clients can only have one packet in flight to the server, which engenders the need for combining multiple operations in a single packet as seen above. This has the implication that as latency increases, the number of packets sent decreases and packet size increases. Clients also must be able to undo operations rejected by the server, as well as transform concurrent, remote operations against local ones. To enable this, clients also buffer past operations. Storing a document of  $n$  characters plus a list of at most  $m$  operations suggests a space complexity  $\Theta(n + m)$ .

### 2.2.3 Convergent Replicated Data Types

This section will provide an intuition for CRDTs for text editing in general. The specific CRDT used for this project is mentioned in §2.2.4 and detailed in §3.1.

CRDTs, first formalized in a 2007 paper [26], trade the complex algorithms used in OT for a more complex data structure. Rather than relying on logic to transform operations against each other, elements are tagged with totally ordered identifiers which allow retrieval of the data in its native form — for example, a string will be represented as a set of tagged characters and may be read out according to the tag ordering. Figure 2.5 is a simple demonstration of how this works.

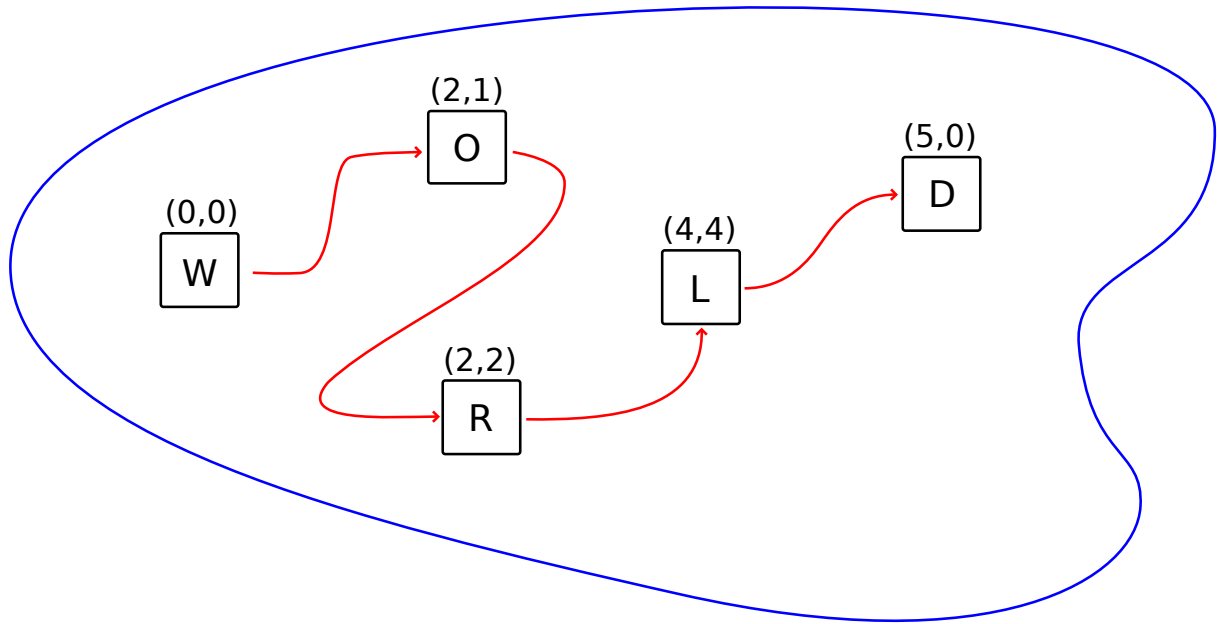


Figure 2.5: A generic CRDT containing the word “World”. The CRDT is a set of nodes tagged with ordered pairs of integers, similar to those used in my CRDT. The order is  $<$ , first applied to the first element of the pair, then the second. The red arrows trace out the ordering, which presents the word “World”.

There are two classes of CRDTs: state- and operation-based. State-based CRDTs disseminate the entire local state to other clients which is then merged into their copies. This requires that the merge operation be commutative, associative, and idempotent [27]. On the other hand, operation-based CRDTs relay modifications to other clients, and require causal delivery between clients, and that concurrent operations commute [30]. This project uses an operation-based CRDT, so these are the key properties to fulfill.

### 2.2.4 CRDTs for Text Editing

I describe a few important CRDTs developed for text editing which demonstrate core ideas used in this project. The RGA CRDT developed by Roh *et al.* [25] is discussed only briefly as it mirrors my own implementation and is detailed further in the next chapter.

#### RGA

RGA [25] stands for replicated growable array. The core ideas developed in the paper are the representation of a character array as a singly-linked list, and the association of each link with a globally unique, totally ordered identifier. The authors use a rather complicated scheme to create these identifiers; the simpler method I develop in §3.1.2 provides the same functionality at lower cost. The paper is quite long and difficult to parse, but ultimately does not offer many practical advances over my independent developments.

The authors also make no mention of undo and redo functionality, for which I extend the CRDT in two different ways.

## Treedoc

Treedoc [23] represents a text buffer as a tree structure in which paths from the root are used to encode identifiers and order elements in the set. Nodes in the tree contain at least one character, and the string contained in the buffer is retrieved using infix traversal. Each client can insert a character tagged with a client ID into a local replica of the tree at any time. Two concurrent inserts at the same node are merged as two ‘mini-nodes’ within one tree node. Infix traversal combined with an ordering over the client identifier creates a total order over the characters contained in the tree, including those grouped as mini-nodes. With the total order, all clients can retrieve the same string from their Treedoc replica. Total orders are an important property used to guarantee eventual consistency in text CRDTs. Using unique, ordered client IDs to break ties in concurrent cases is common in CRDT design and indeed used in my own CRDT.

Deletions are handled by marking nodes as removed rather than removing them from the structure, putting Treedoc into the class of *tombstone* CRDTs.

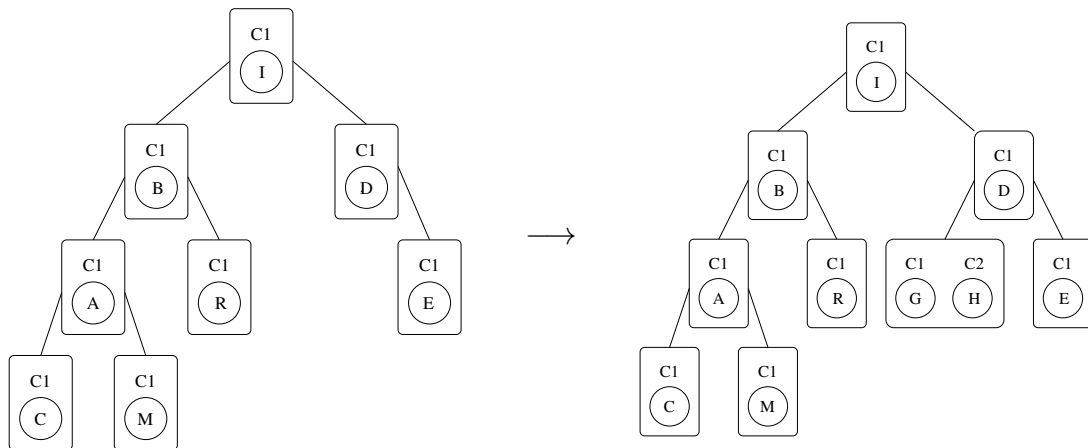


Figure 2.6: Concurrent updates to the same node. In the first state (left), user 1 has written the string ‘Cambridge’ into the text buffer. The systems settles and user 2 also sees the string ‘Cambridge’. Both users 1 and 2 realize there is a missing character — user 1 inserts ‘g’ and user 2 mistakenly inserts ‘h’. As both create the same node concurrently, they are merged as mini-nodes in the same tree position. Both users now see the string ‘Cambridghe’.

## Logoot

Logoot [34] belongs to the class of text CRDTs that do not require tombstones for deletion. It achieves this by creating totally ordered identifiers without implicit dependencies on other parts of the CRDT (for example, Treedoc nodes depend on the existence of higher

nodes in the tree). This means that to delete, any client can simply remove an identifier and the data it tags without adverse effects on the rest of the CRDT.

The process of generating identifiers independently of each other is an expensive process and results in long identifiers. However, the lack of tombstones and improvements made in two further papers [19] [18] provide distinct advantages of prior CRDTs such as Treedoc. Relevant is the idea that identifier generation can be expensive in terms of both space and time: this project uses a simpler identifier scheme.

### Logoot-Undo

CRDTs generally struggle to provide an undo mechanism since the concept of reversing an update to the data structure breaks causal dependencies. As stated above, Logoot does not embed such dependencies in the CRDT, so implementing an undo and redo as deletion and insertion of a new character and identifier would work. However, this mechanism fails in a concurrent case: two users delete a character concurrently, then undo its insertion independently. The character would appear twice, which is incorrect.

Logoot Undo [33] solves this by essentially tagging each character with a *visible* counter. An undo of an insertion decrements it, while redo increments it. Only if the counter is positive is the character shown. As discussed in §3.4.3, this leads to some rather unexpected behavior. However, this approach is viable since increments and decrements commute and guarantee eventual convergence. The use of a counter is identical to one undo mechanism I developed independently, though I chose to implement a local undo, only affecting locally generated changes, rather than a global one presented in the paper, which allows any client to undo any other clients' operations.

## 2.3 Starting Point

As stated in the proposal, I had prior experience with ShareJS, which was used as the basis of the comparison system. Additionally, I already had some knowledge of Typescript, my main implementation language. However, almost all other aspects of the project were new to me.

As the project progressed, several courses contributed or reaffirmed ideas I could use. Notably, the Computer Systems Modeling [7] course had a section on simulation which aligned very well with what I had already implemented at the time. The Part IB course on Concurrent and Distributed Systems [32] provided valuable background towards Lamport and Vector clocks, causality, and total orderings in distributed systems; the IB Computer Networking course [6] gave me a solid foundation for planning the network component of my system.

## 2.4 Requirements Analysis

To reiterate the success criteria listed in the project proposal, I hoped to

1. Implement a concurrent, distributed text editor based on CRDTs
2. Pass correctness tests for this CRDT
3. Obtain and compare quantitative results from ShareJS- and CRDT-based systems

Points one and three have multiple subgoals. For clarity, Table 2.1 lists these and their respective importance and difficulty. The completed extension is included as well.

Table 2.1: Project (Sub)goals, priority, and difficulty

Goal	Priority	Difficulty
Implement and unit test core CRDT	High	Medium
Implement network simulation	High	High
Optimize CRDT Insert	Low	Low
Design experiment format	High	Low
Create comparative ShareJS system	High	Medium
Write log analysis scripts	Medium	Low
Extension: Undo/Redo capability	Low	High

## 2.5 Software Engineering

### 2.5.1 Libraries

ShareJS [12] v0.6.3 (MIT licensed) is the main external resource I required. This package was installed via the NPM<sup>3</sup> package manager. The other external library I used was D3.js<sup>4</sup>, a data visualization tool that helped me build a dynamic network graph for debugging purposes.

---

<sup>3</sup><https://www.npmjs.com/>

<sup>4</sup><https://d3js.org/>

### 2.5.2 Languages and Tooling

The three main implementation languages, by lines of code, are Typescript<sup>5</sup>, Python 2.7<sup>6</sup>, and Coffeescript/Javascript (mainly in ShareJS). I chose Typescript as my primary implementation language because of familiarity, how easily it integrates with web technologies and JSON objects, static typing (which helps greatly with project scale and early error detection), and the fact that ShareJS ships as Javascript, which Typescript transpiles to. My development stack included package manager NPM, Typescript, transpiler Babel, and script bundler Webpack; coding was done in Visual Studio Code, an open source IDE developed in parallel with Typescript by Microsoft. How to couple all these tools together correctly is an issue in itself, and setting up a working configuration was one of the most tedious preparation steps.

In order to maximize code reuse and comparability of results both systems must be evaluated on the same platform, a web browser. The most developer friendly choices are Mozilla Firefox<sup>7</sup> and Google Chrome<sup>8</sup>, as both come with sophisticated debuggers and script inspection capabilities. I chose Google Chrome as it offers a simple API to obtain heap usage, and its shortcomings in terms of a limited number of active TCP connections are solved with a workaround built into ShareJS.

### 2.5.3 Backup Strategy and Development Machine

Github<sup>9</sup> provided the primary backup, with Dropbox also continually backed up to the cloud. To prevent data loss in event of operating system failure, the primary development OS Ubuntu 14.04 LTS x64, backup OS Windows 10, and user data each reside on their own hard drives. The MCS computers were the alternative in case of loss of laptop.

## 2.6 Early Design Decisions

### 2.6.1 Network Simulation

I originally chose to implement a network simulation for the CRDT-based system as it allows me to control the topology, link latencies, and protocol and to explore advantages and disadvantages of using peer-to-peer versus a client-server model (especially the role of the degree of connectedness in a P2P network). In addition, I had very little experience with P2P network and associated routing algorithms and thought it more worthwhile to learn simulation techniques than delve into networking.

---

<sup>5</sup><http://www.typescriptlang.org/>

<sup>6</sup><https://www.python.org/>

<sup>7</sup><https://www.mozilla.org/en-US/firefox/new/>

<sup>8</sup><https://www.google.com/chrome/>

<sup>9</sup>[github.com](https://github.com)



However, I also had no experience with simulations, so I simplified wherever possible. My system assumes the network guarantees in-order delivery and is capable of a broadcast to all peers of a node<sup>10</sup>. Note that broadcast, or flooding, has severe implications in terms of network efficiency. Without further measures, it sends  $\Theta(n^2)$  packets in a fully connected network, where  $n$  is the number of clients. However, though it has downsides, I use broadcast because of its ease of use with any network topology, and lack of addressing and sophisticated protocols. In §3.1.3 I will relax the need for in-order delivery using vector clocks.

### 2.6.2 Data Collection and Logging

Other important design decisions are more general. One is to measure all packet and data structure sizes in terms of number of characters they require when converted into strings using a standard JSON object to string conversion. This allows fair comparisons between my system and ShareJS, and is the most obvious way to measure the size of a JSON object. Alternatives exist which provide more efficient serializations, such as Protocol Buffers [15]. However, utilizing a more efficient scheme than ShareJS would make for unfair comparisons.

The second decision is to log network packets on the application layer. That is, rather than intercepting and logging packet information at the operating system level, I log payloads of packets from within the applications. This makes comparisons between network traffic more fair, as one system's packets contain no headers, and the other's have TCP/IP overheads.

---

<sup>10</sup>True broadcast is not typically found in Internet applications. For instance, IPv6 phases out broadcast functionality and opts for multicast instead [5]



# Chapter 3

## Implementation

This chapter describes the implementation of two real time editing systems, firstly one based on CRDTs, and secondly one based on ShareJS. Following is an overview of the experiment generation and analysis components that are shared by both systems. Lastly the extension that adds Local Undo capability to the CRDT is discussed.

### 3.1 CRDT-based system

#### 3.1.1 Overview

The high level components that make up my CRDT-based text editor are the user interface, the CRDT, and the network simulation. Each simulated client owns a local replica of the CRDT, an editable text area, and a simplified network stack. The stack disguises from the client the simulated network I created to transmit packets between clients. This approach aims to ease exchanging the simulation for a P2P protocol such as WebRTC<sup>1</sup> at a later date. It also allows separate development of the networking subsystem and the CRDT. Such separation of concerns and independence between subsystems are core principles of software engineering that were adhered to throughout the implementation. Figure 3.1 shows the architecture of the whole system.

Upon interaction with a client's text interface, the CRDT is modified and the operations generated are passed via the network to other clients. Upon receipt, the remote clients integrate the changes into their replicas of the CRDT and update the user interface to reflect the new state. The following sections describe the generation and integration of operations.

---

<sup>1</sup><https://webrtc.org>

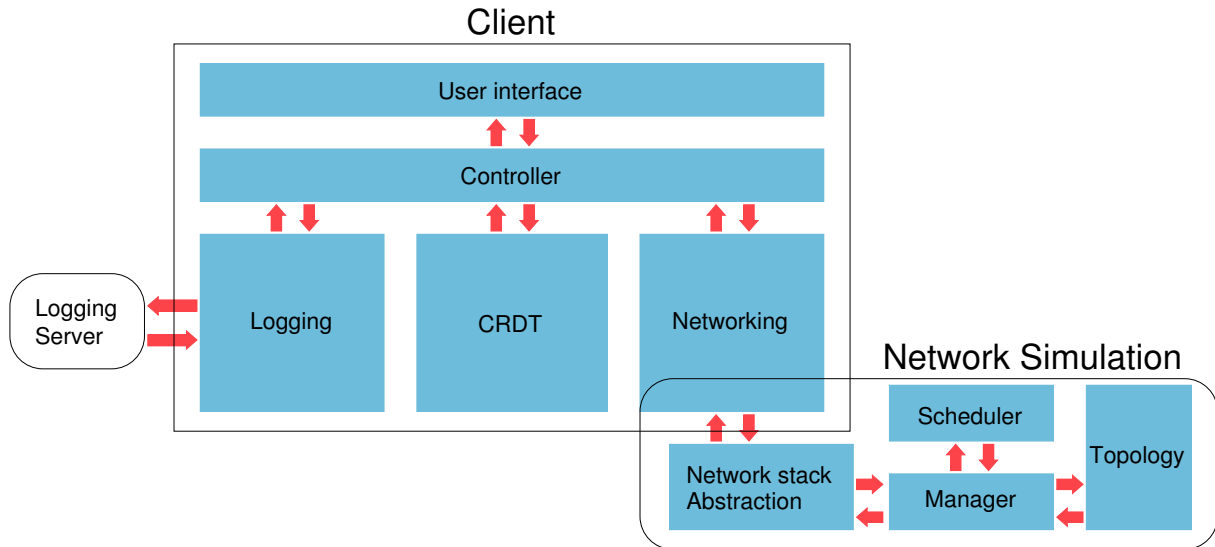


Figure 3.1: An overview of the system architecture implemented for the CRDT-based text editor. Each client has its own UI, controller, CRDT, logger, and networking component. The network module abstracts away the underlying simulation. Many clients can connect to the network abstraction and communicate through it.

### 3.1.2 CRDT

This subsection goes through the structure and capabilities of the CRDT I utilized, how character identifiers are generated and totally ordered, the operations that are supported in the context of text editing, a brief discussion of tombstones left behind by deletions, and optimizations.

#### Structure and Functionality

Previously in §2.2.4 various CRDTs were mentioned, such as Treedoc which stores characters in the nodes of a tree and retrieves them in infix order. Rather than using a tree to store characters, my approach utilizes a singly-linked list. Each link contains a character  $c$ , a pointer  $n$ , and is associated with a unique identifier.

A CRDT needs to implement three core methods in order to support text editing

1. **Insert:** Add a character at a specific index or location
2. **Delete:** Remove or mark as deleted any given character
3. **Read:** Retrieve the characters in order and return them as a string

Various data structures were considered when implementing the linked list. The intuitive approach is to implement each link as an instance of a class or structure containing the required identifier, character, and pointer. However, all operations one might want to do on such a linked list are  $\Theta(n)$ : finding a node to insert after or delete requires scanning some proportion of the list.

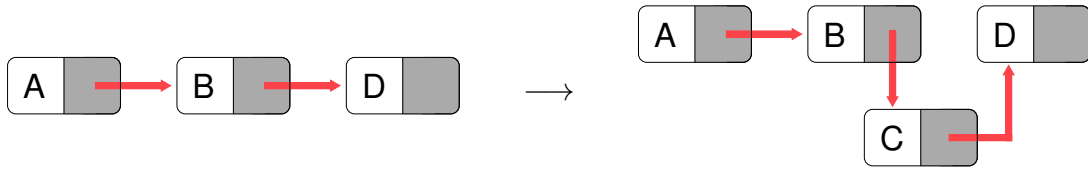


Figure 3.2: A graphical representation of updating linked lists. Here, a new link containing the character ‘c’ is inserted between ‘b’ and ‘d’ to produce ‘abcd’.

A better approach is to implement the linked list within a hash table. Since node identifiers are required to be unique, they can be used as keys in the map and achieve close to constant lookup times and thus fast insert and delete operations. To be more precise, under the assumption of uniform hashing, a hash function is  $O(m)$  where  $m$  is the length of the key. The keys used in my CRDT are of length  $\Theta(\log(n) + \log(r))$  where  $n$  is the number of characters in the CRDT and  $r$  is the number of replicas in the distributed system. Figure 3.3 gives a sample CRDT implemented within a Javascript Object hash map.

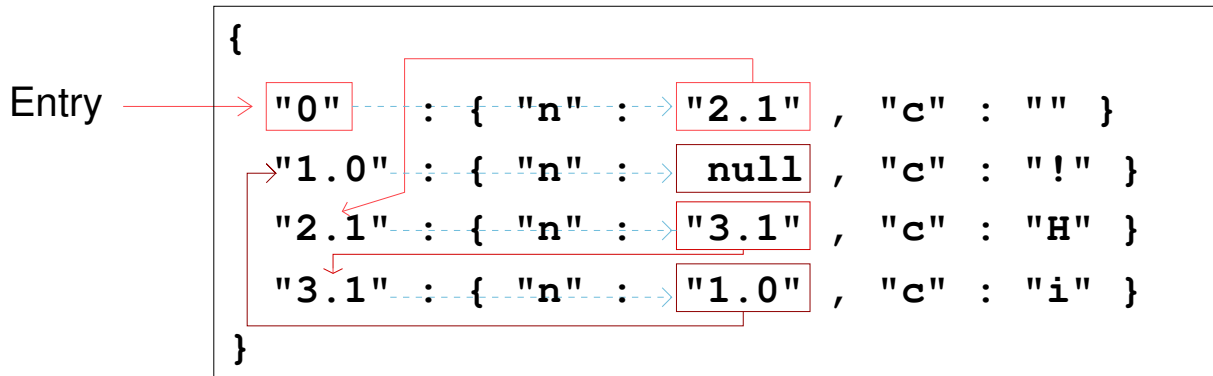


Figure 3.3: A sample CRDT annotated with arrows. The red arrows denote the links in the linked list. The invariant anchor element tagged ‘0’ is always retrieved first and invisible to the user. The final link, tagged “1.0” has no further pointer. Pointers are labeled *n* in the diagram. The string contained in this instance is “Hi!”.

Javascript provides two native objects capable of mapping: **Map** (supports arbitrary objects as keys by hashing pointers) and the standard Javascript **Object** (only allows string and integer keys). I implemented my CRDT using both structures and eventually settled on **Object** for its ease of serialization to JSON and the inconvenience of using **Map**: hashing pointers to keys, rather than the key contents, makes it difficult to work with if pointers are not retained throughout the lifetime of the program.

## Identifiers

Recall that CRDT identifiers are required to be globally unique and totally ordered. My CRDT is advantageous over tree-based CRDTs in that generating identifiers is straightforward. Each client has a unique ID (referred to as *cid*) which forms one part of each

identifier. A *cid* can either be randomly generated or provided by the bootstrapping server for the P2P network. In this project, the network simulation provides unique *cids*.

Define an identifier generated by client *i* to be a pair  $(t_i, cid_i)$  where  $t_i$  is the value of a local counter incremented on every insert.  $cid_i$  is the globally unique identifier of *i*. Since each client provides a monotonically increasing *t* per identifier, and *cid* is globally unique, every pair generated in the system is guaranteed to be unique.

The counter  $t_i$  is maintained as a Lamport clock [17]. This means that when receiving an incoming operation with a greater  $t_j$  than the local clock, set the value of  $t_i$  is set to  $t_j$ . This guarantees that only concurrent operations can have the same clock value, and is useful when incorporating a remote insert operation into a local CRDT replica.

I now define a total order  $<_{id}$  over the identifiers:

$$(t_1, cid_1) <_{id} (t_2, cid_2) \Leftrightarrow t_1 < t_2 \vee (t_1 = t_2 \wedge cid_1 < cid_2)$$

## Insert Operation

Insertion into a text document is one of the two fundamental modifying operations that must be sent to other clients. An insert operation is stored and transmitted as an insert bundle containing:

- A unique identifier **id** for the character
- The character **char**
- The unique identifier **after** of the node after which to insert the character. This corresponds to the position in the linked list at which the character needs to be spliced in.

```

1 interface InsertMessage {
2     id: string,
3     char: string,
4     after: string
5 }
```

Listing 3.1: Insert Bundle Type Signature

Incorporating an insert bundle B into the CRDT acts according to the following pseudocode.

```

1 prevNode = map.get(B.after)
2 while prevNode.next >_{id} B.id do
3     prevNode = map.get(prevNode.next)
4 map.add(B.id, Node(B.char, prevNode.next))
5 prevNode.next = B.id
```

Listing 3.2: Incorporating Insert Bundle into CRDT

This is the standard procedure to insert a new node into a linked list, with the exception of lines 2 and 3. These are the key steps to ensure that all clients converge to the same string, no matter the order the concurrent inserts are incorporated (commutativity property). Figure 3.4 demonstrates the effect of the **while** loop: it is enforcing the total ordering defined previously. This ensures all clients incorporate concurrent operations in the same position and hence converge to the same result. The stopping condition ( $\neq_{id}$ ) is enforced by the use of Lamport clocks for identifiers.

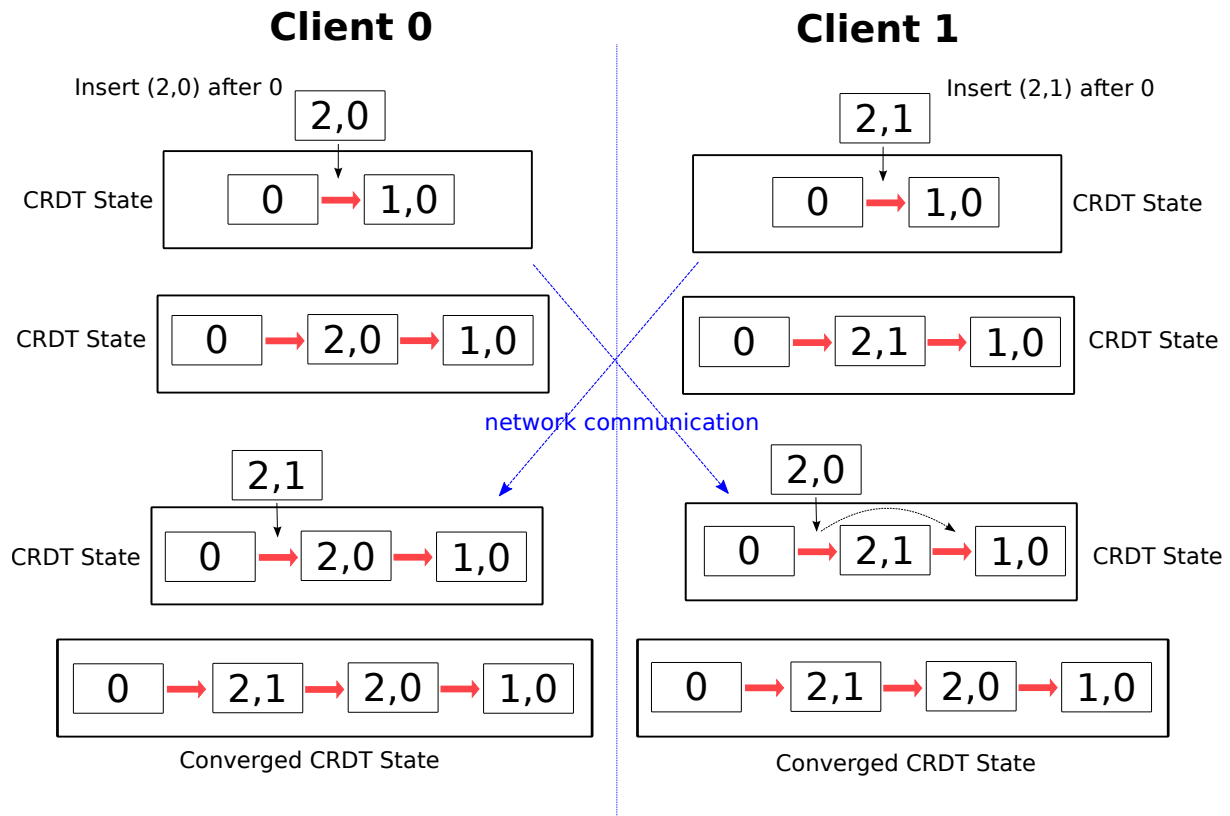


Figure 3.4: This simplified figure showing only character identifiers demonstrates the use of the total ordering to converge to an identical result on all clients. Two clients concurrently insert after identifier 0. After exchanging operations, Client 1 inserts the remote 2,0 identifier after its own 2,1 according to the total order defined, while Client 0 inserts 2,1 ahead of its locally generated 2,0.

**Delete** Deleting a character from the text document generates a delete operation, which is transmitted as a bundle containing

- The target character's identifier to be deleted `deleteId`

```
1 interface DeleteMessage {
2     deleteId: string
3 }
```

Listing 3.3: Delete Bundle Type Signature

Incorporating a delete bundle B into the CRDT is straightforward

1. Locate the node  $N$  corresponding to  $B.deleteId$
2. Set a boolean flag  $N.d$  to `true`

## Tombstones

The delete operation described previously never removes nodes, but leaves them behind as tombstones. Some CRDTs, such as Logoot described in §2.2.4, are structured such that the document is a series of independent nodes, which are arranged solely according to their identifiers. Thus, a node can be fully removed without consequence to other nodes. In my CRDT, each node depends on the prior node in the linked list. Unless we can establish that each client has received and executed a delete operation, we cannot remove a node from the linked list. In general this cannot be assumed as other clients may be executing concurrent edits which depend on that node, are working on a document offline, or on a high latency link.

The process of establishing that each client has received and executed an operation can be achieved using an expensive commitment protocol, as suggested in [23]. In effect, the system periodically executes a distributed garbage collection. Removing tombstones may be necessary when the data structure becomes unwieldy, until such a point tombstones are useful in implementing undo functionality for the text editor. This is discussed in section §3.4. I also discuss an alternative garbage collection scheme in §3.4.3.

## Optimizations

The insert operation produces a bundle which contains exactly one character, one identifier, and the identifier of the node to insert after. The number of operations generated can be cut drastically by allowing an insert bundle to contain a contiguous sequence of characters and a single identifier.

An optimized insert bundle contains

- A unique identifier  $(t, cid)$  for the first character
- The character sequence  $s$  itself
- The node after which to insert the character sequence denoted  $(t_{after}, c)$

The receiving CRDT incorporates the bundle by generating a new node for each character  $s_i$  with identifier  $(t + i, cid)$ . The first node is pointed to by  $(t_{after}, c)$ 's *after* pointer and each new node points to  $(t + (i + 1), c)$ . The final node has a pointer to the original target of  $(t_{after}, c)$ . The resulting size of the CRDT is the same as when using multiple unoptimized bundles: this optimization only improves network efficiency.

With the optimization, at best, an entire document could be inserted at once, sending exactly one identifier plus a string of length  $n$  in a single packet. A more likely scenario is word-by-word or line-by-line insertion. At worst, we revert to the unoptimized case:



$n$  characters and identifiers are sent in  $n$  packets. If we assume the network can send arbitrarily long packets, the application-layer network capacity requirement is  $\Theta((m + \log(n)) * n/m) = \Theta(n + n \log(n)/m)$ , where  $m$  is the average number of characters sent per packet. Written another way this is reduced to between  $\Omega(n)$  and  $O(n \log(n))$  rather than  $\Theta(n \log(n))$ . The number of packets sent is reduced to  $\Theta(n/m)$ , which is desirable since standard protocols such as TCP have high per-packet dissemination overheads.

Another optimization was the renaming of tags and names to be as short as possible (often single characters), shortening the resulting serialized JSON string sent over the network. As discussed before in §2.6.2, alternatives to JSON such as Protocol Buffers would eliminate almost all of these overheads, but this would make comparison with ShareJS less direct.

### 3.1.3 Simulated Network

This section describes the simulated network I created to deliver operations from one client to another. First, I will detail the initial assumptions made. This is followed by the abstractions the simulation provides to each client. Next I describe the core difficulty in implementing a simulation: the scheduler. Lastly, I will relax the assumptions made below to more closely mirror real world situations.

Note that the term *simulation* is not exactly correct and *emulation* might be slightly more accurate. However, for simplicity and consistency, I will use *simulation*.

#### Assumptions

The CRDT I implemented, as it is an operation-based CRDT not a state-based one, requires that messages be delivered causally. We define the happens-before relation  $a \rightarrow b$  to be true whenever  $a$  happens before  $b$  on the same process, for example

Receive Insert “*Hello*”  $\rightarrow$  Insert “*World*” after “*Hello*”

We then require that all events ordered by  $\rightarrow$  be delivered in a valid ordering according to  $\rightarrow$ . This is called *Causal Delivery*.  $\rightarrow$  is a partial order, since there may be some  $A$  and  $B$  such that neither  $A \rightarrow B$  nor  $B \rightarrow A$  holds. Concurrent operations can be delivered in any order since they are guaranteed to commute by the properties of the CRDT.

To justify the causal order delivery requirement, simply take the case of inserting a link into a linked list after a node that does not exist yet: potentially causal operations must be delivered in order.

Network Assumptions. These are guaranteed by the simulation.

1. Unchanging network topology
2. In-order delivery on any single link in the network

### 3. No packets are lost

Along with this, my implementation of individual clients guarantees:

1. Received packets are forwarded in order, i.e. if  $A$  arrives before  $B$ , then  $A$  is forwarded before  $B$ .
2. Received packets are flooded to peers before the client generates and broadcasts potentially dependent operations.

The simulation and client guarantees provide causal delivery.

*Proof of Causal Delivery Guarantee.* Assume there is some packet  $A$  in the network. A packet  $B$  is sent in a causally dependent manner, i.e.  $A \rightarrow B$ , thus  $B$  must be delivered after  $A$  on every client in the network. Denote the network node which generated  $B$  as  $sender_B$ .

Proof by induction along any shortest path  $p$  from  $sender_B$ , with induction variable  $i$ , where  $p_i$  is the  $i^{\text{th}}$  hop on  $p$  from  $sender_B$ . The flooding mechanism used in the simulation first delivers packets to every node via the shortest path from a sender as long as the network is static and no packets are lost, which are guaranteed by network assumptions 1 and 3.

Base case,  $i = 1$ .

As  $p_1$  is exactly one hop from  $sender_B$ , and  $sender_B$  must have put the packets onto the link in order by client assumption 2, and packets are delivered in order over any link,  $p_1$  must receive the packets in order.

Inductive case,  $i = m$ .

Assume  $p_m$  receives packet  $A$ , then  $B$  i.e. in order. By client assumption 1 (in-order forwarding), and network assumption 2 (in-order delivery on individual links), node  $p_{m+1}$  must receive  $A$  followed by  $B$ . Because  $p_{m+1}$  also lies on the shortest path, this must be the first delivery to  $p_{m+1}$ .

This holds over every shortest path through the network, including over all identically long paths to a node. Thus, the network guarantees causally ordered delivery to every node in the network.

□

That the network is able to guarantee causal delivery is a strong assumption, and cannot be made in reality. I will show in §3.1.3 how to relax network assumptions 1 and 2.

## Upper Network Abstraction: Network Interface

My network simulation is implemented in two parts: a manager which is shared between all simulated clients, and a **Network Interface**, of which each client has an instance. The **Network Interface**, whose type signature is shown below, essentially replaces the top of a classic network stack, while the manager abstracts away the bottom layers.

```

1 interface NetworkInterface {
2     isEnabled: () => boolean;
3     enable: () => void;
4     setClientId: (ClientId) => void;
5     setManager: (NetworkManager) => void;
6     requestCRDT: (ClientId) => void;
7     returnCRDT: (ClientId, MapCRDTStore) => void;
8     broadcast: (PreparedPacket) => void;
9     receive: (NetworkPacket) => void;
10 }

```

Listing 3.4: NetworkInterface Type Signature

The primary packet dissemination method is via the `NetworkInterface.broadcast` method. It broadcasts a `PreparedPacket`, which contains a bundle and a tag to disambiguate the type of bundle, as type information is lost during serialization over the network.

```

1 interface PreparedPacket {
2     type: "i" | "d" | "reqCRDT" | "retCRDT", // insert or delete
        message, or request/return CRDT
3     bundle: CRDTTypes.InsertMessage | CRDTTypes.DeleteMessage |
        RequestCRDTMessage | ReturnCRDTMessage;
4 }

```

As expected, the bundles are either insert or delete operations. Two other types of bundles that can be sent are `RequestCRDTMessage` and `ReturnCRDTMessage`, special messages which clients use when joining the network.

### Lower Network Abstraction: Network Manager

The lower layers of the network stack are provided by the `Network Manager`. It has two key methods: `NetworkManager.transmit(sender, packet)` and `NetworkManager.unicast(from, to, packet)`. The simulation is given a predefined topology, which contains connectivity and latency information. Thus, when a client's `NetworkInterface` calls `NetworkManager.broadcast`, the manager knows which clients are neighbors and corresponding link latencies, and can schedule a delivery event for each. The `NetworkManager.unicast` is used for point to point, single hop communication when joining the network and requesting or returning copies of CRDTs. This module does the work of the network and data layers of traditional network stacks and abstracts away how packets travel between neighboring clients.

### Joining the Network

To make the system more flexible, I added the capability to join the P2P network during simulation. Only the first client gets to create a new CRDT; joining clients request a copy of the CRDT via `NetworkInterface.RequestCRDTMessage` (see §3.1.3).

An alternative method to requesting an up-to-date CRDT is to begin with an empty CRDT and replay all subsequent operations on it. This would however be significantly less efficient as it requires all remote clients to store all of their previous operations forever, and joining clients would each have to reintegrate all operations. On the other hand, doing a partial state replay would be simpler: transmit only the missing operations. This is not easily done in my implementation. If a client has an out of date CRDT, it must request an entire new copy via `NetworkInterface.RequestCRDTMessage`.

At this point it is important to acknowledge that introducing dynamic network joining violates one of the guarantees of §3.1.3. Namely, incorporating new clients over time changes the network topology and thus the guarantee of causal delivery no longer holds. The introduction of vector clocks below will restore this guarantee.

### Causal Delivery

Until this point, the network has guaranteed causal delivery of packets based on strong assumptions and knowledge of the system implementation. These can be relaxed to allow out of order delivery and dynamic network topologies. This can be done using vector clocks [11].

I introduce an additional network layer, as depicted in Figure 3.5. I make a distinction between receiving and delivering a message. Receiving is the arrival of a message at a client, whereas delivery passes the message up the network stack. Causal delivery guarantees that messages are delivered such that  $A \rightarrow B \Rightarrow \text{deliver}(A), \text{deliver}(B)$ .

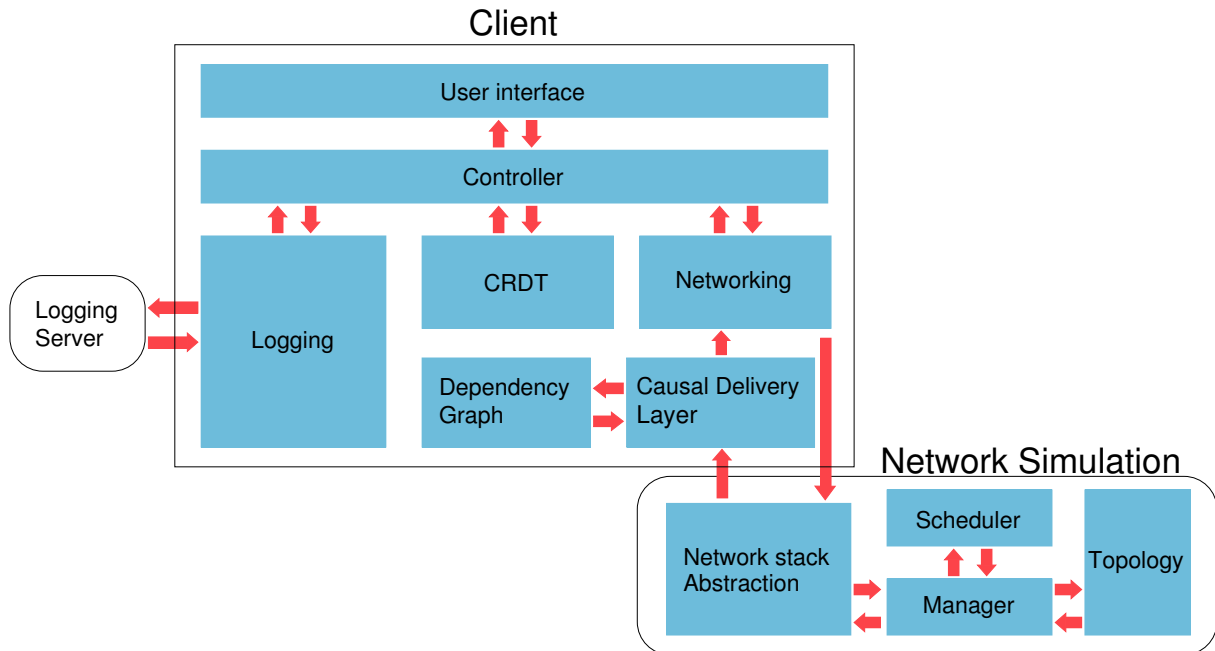


Figure 3.5: The updated architecture depicting the additional network layer ensuring causal delivery.

**Modified Vector Clocks** A traditional vector clock is outlined in Appendix A.1. In this system, the causal delivery layer must delay messages that arrive before all of their causal dependencies are delivered, which requires calculating exactly which messages are missing. To this end, I modified the vector merge rule (number 4 from the appendix) to only increment the local clock whenever a message is generated locally. This way, each value in a vector represents exactly how many messages are known to have been sent by the corresponding client. The full modified rules are listed in Appendix A.2.

We can now determine concurrent, causally dependent and causally prior messages by comparing vectors. We define the latter two about  $<_v$ .

Two vectors  $v_1$  and  $v_2$  occurred concurrently iff

$$\exists c, c' \in v_1, v_2. v_1.c > v_2.c \wedge v_1.c' > v_2.c'$$

Given vectors  $v_1$  and  $v_2$  with components  $v.c$ , define

$$v_1 <_v v_2 \iff \forall c \in v_2, v_2. v_1.c \leq v_2.c \wedge \exists c' \in v_1, v_2. v_1.c' < v_2.c'$$

We say  $v_2$  is *causally dependent* on  $v_1$  and  $v_1$  is *causally prior* to  $v_2$ .

If a client receives a vector  $v$  that is concurrent with the client's current vector  $s$ , the causal network layer immediately delivers the message to the client. If  $v <_v s$ , i.e. is causally prior, then the message has been seen before and can be discarded. If  $s <_v v$ , i.e. is causally dependent and not seen before, a delta can be computed by taking the element-wise difference between  $v$  and  $s$ , treating missing elements as 0. This represents the number of messages missing from each client.

Calculating missing messages is useful in delivering buffered messages. The naïve approach is to scan the entire buffer on every packet delivered up the stack, which can result in a cascade of deliveries at quadratic cost. The delta enables building a dependency graph and only checking neighbors of delivered packets for further deliveries.

### 3.1.4 Simulation

A large part of the infrastructure underlying the CRDT is the simulation driving the network and executing preprogrammed events. The core of this is the simulation scheduler which I focus on below. Other components are interfaces to the experimental setup and networking modules.

#### Scheduler

A simulation scheduler is responsible for mutating system state based on events to be executed at specific times. To schedule an event, an object needs to call the `Scheduler.addEvent` method, listed below.

```

1 public addEvent(time: number, clock: number, action: any) {
2   let heapElem: DualKeyHeapElement = {
3     pKey: time,
4     sKey: clock,
5     payload: action
6   };
7   this.heap.insert(heapElem);
8 }

```

Listing 3.5: The Scheduler.addEvent method

The scheduler uses an underlying heap to manage events, which makes no first-in first-out guarantees. As it is possible for a client to submit multiple packets onto a single link at the same logical time, and §3.1.3 requires in-order delivery on any link, my heap uses a secondary key *sKey* to order delivery.

The key property of a correct scheduler, as noted in the Part II Computer Systems Modeling [7, slide 120] course, is that the next executed event be the one with the least remaining time. Using a heap, we get  $\Theta(\log(n))$  time per element retrieved. Executing an event may generate more events which are added back into the heap.

The scheduler is seeded with events defined in an experimental setup, see §3.3.

## Representation of Time

The concept of time in a simulation is generally taken to be “logical time”. The system begins at  $t = 0$ , and each subsequent event moves the  $t$  variable forward. This works perfectly well for the network simulation, since the latency on any individual network link is well defined and deterministic.

The alternative ‘time’ that can be used is ‘wall-clock’ time. The amount of time until some next event is given in milliseconds to wait, rather than a logical delta which is skipped over. Using this concept of time in a simulation introduces extra complexity, primarily stemming from inaccuracy in timers provided by the host platform. Indeed, it is likely that some events would have very small deltas, for which starting and stopping a timer would be nearly impossible.

In this project, I implemented both an event-driven scheduler and a timer-driven scheduler. The timer-driven version is useful for debugging and watching the simulation unfold in real time, whereas the event-driven version runs as fast as the hardware permits. However, I found that I could, to an extent, emulate the timer-driven scheduler using the event-driven scheduler by adding a sleep proportional to the  $\Delta t$  until the next event. As the event-driven version is more flexible, and simpler – the driver is a simple while loop, rather than recursively set timers with callbacks – I decided to use it when executing experiments on the CRDT-based system.

The timer-driven scheduler is still useful in the comparative system (§3.2), as packet deliveries are nondeterministic.

## 3.2 ShareJS Comparative Environment

The main difficulty when building the ShareJS-based comparative system was adapting it to allow executing the corresponding experiments to the ones executed in the CRDT-based system. This meant incorporating the real time scheduler discussed in §3.1.4, and inserting log statements to record data to the logging server.

### 3.2.1 Scheduling

An experiment consists, at the most basic level, of a set of simulated events that each client performs at specified times. ShareJS requires that sockets be used to transmit operations between clients, even if they are all on the same machine, which introduces nondeterminism and necessitates using the real time scheduler rather than the logical time one. The scheduler is seeded according to an experimental setup, inserting and deleting characters via hooks available in ShareJS.

The scheduler, logging module, and various helper functions were reused exactly as in the prior system. Their use led to a Typescript/Javascript combined system, but since Typescript is fully compatible with Javascript no major difficulties were encountered, except the lack of type annotations in ShareJS.

### 3.2.2 Logging

Enabling data logging primarily consists of inserting log lines at critical points that reveal interesting information about the system. The most important of these are the receiving or sending of any packets from a client or the server, and total memory consumption of the clients before and after the experiment. This was done by reading the ShareJS source code and inserting references to a global logging module at the appropriate points. The server was also modified to log relevant information.

## 3.3 Experiment Creation and Use

This section deals with the creation and analysis of the experiments. First I outline the overall system work flow. Then, I discuss the design of an individual experiment. Lastly, I briefly examine the decision to log experiments to text files, then separately parse and analyze these files to collect data.

### 3.3.1 Work Flow

On the whole, this project operates as in Figure 3.6.

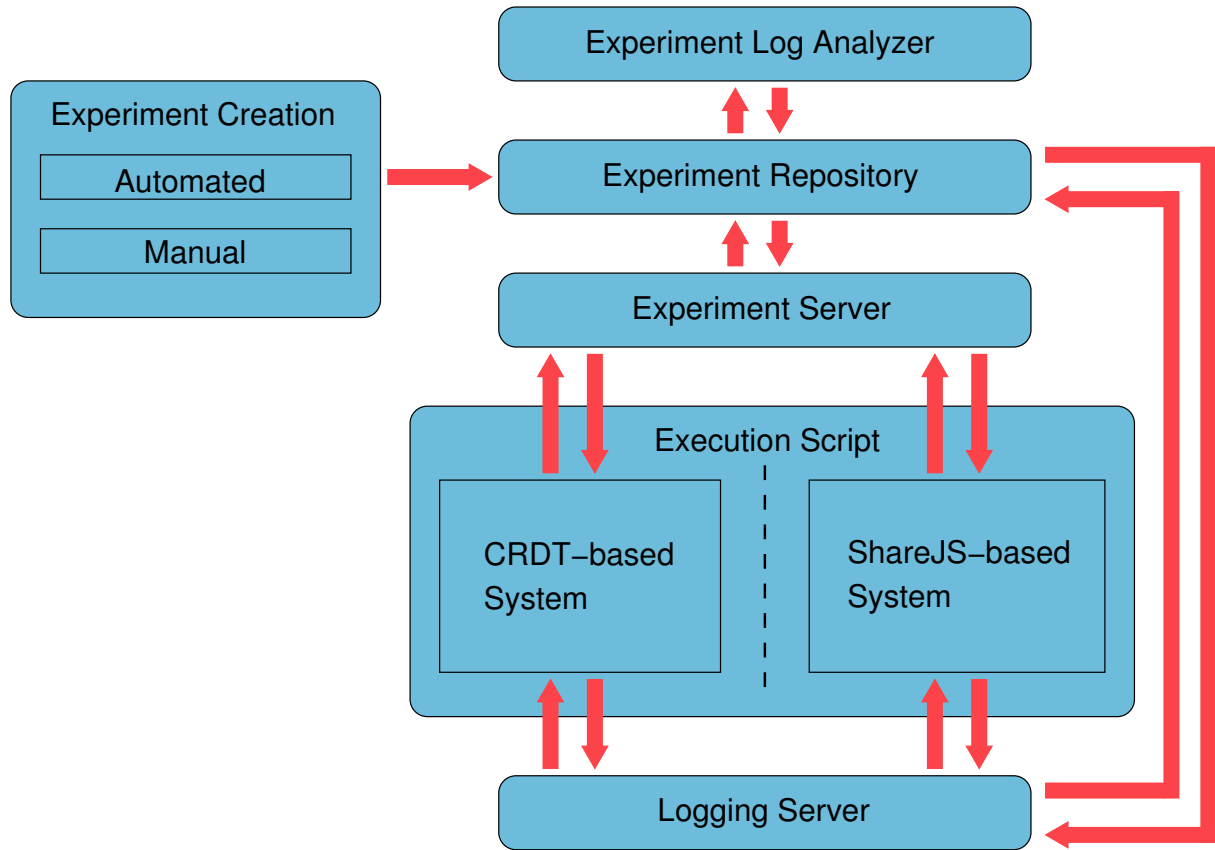


Figure 3.6: The standard work flow for creating, running, and analyzing experiments.

Manually or with a custom script, experiment setup files are created, with specifications over which variations should be executed (such as different topologies or optimizations enabled). An experiment server provides whichever experiment is queued to the requester, either the CRDT-based system or the ShareJS-based one. In either case, the same basic data is served. The system runs the experiment and submits the resulting log back to a logging server. At some future point, a separate script collects logs and summarizes them into digestible formats. Between experiments, both the testing platform (Google Chrome) and the ShareJS server, if applicable, are relaunched.

### 3.3.2 Experiment Design

A basic experiment must define the number of clients participating, a set of events for each client to execute, and the network topology. The events are seeded in the scheduler during an execution. Appendix C.1 shows a sample experiment setup in full.

I decided to assign latencies per-node, and calculate a link latency as the average between two nodes' values, rather than assign per-link because of need to have one network description that fits any topology. This also models the real world better: a device with one high latency link, such as a mobile phone, will likely have only higher latency links rather than some fast and slow links. This "higher on all connections" is modeled with an average between a (high-valued) node and its neighbors.



### 3.3.3 Separation of Modeling and Analysis

Modules for tracing logged packet departures and arrivals end up resembling a set of ‘clients’ and would have fit well into the systems at runtime. However, I decided that having a shared layer that is easily extensible is worth some high level duplication. Another reason is that it is good software engineering: one component should do one task and data creation and analysis are very separate objectives.

An independently running Python script serves this purpose. A sample analysis from a simple experiment is listed in Appendix C.2.

## 3.4 Extension: Local Undo

In interactive systems undo and redo are key features [28]. As such, these are interesting extensions to the capabilities of a CRDT for text editing beyond basic insertion and deletion. I chose to implement a local undo — that is, only allow undoing and redoing operations that were performed locally — rather than a global undo, where anyone can modify any operation. This is because global undo quickly becomes confusing and counterintuitive during concurrent editing: performing an undo may not affect the user’s last operation but unexpectedly revert someone else’s. The approaches detailed below were developed originally; one is similar to an implementation I found at a later time in Logoot Undo (§2.2.4).

### 3.4.1 Overview

Each client keeps a local stack of operations that it produced. The stack can be truncated to avoid ever-growing memory consumption. To undo an operation, the stack pointer is moved down the stack and the inverse operation of the corresponding item on the stack is generated, executed, and broadcast to other clients.

There are two operations that can be undone or redone: insert and delete. Only one client can ever locally generate an insert event (as each character is tagged with a globally unique identifier generated on the client) and thus only it can undo and redo the operation. However, multiple clients can concurrently delete the same character and so have the same `delete(node)` operation on their local stack. How this concurrent case is handled leads to two different semantics and consistency models.

### 3.4.2 Undo and Redo of Insertions

Undo and redo of an insert are not equivalent to deletion and insertion, due to the unique identifiers generated during insertion events. Rather, to satisfy the widely used CCI

semantics, a redo must reintroduce the same character with the same unique identifier. The solution to this is discussed in the following subsections.

### Undo/Redo Insert Semantics

To satisfy the CCI consistency model and semantics (§2.1.2), an undo and a redo should have *the same effect on the visible text as if the prior creation or undo never occurred*. For instance, if  $A$  creates a character which is deleted by  $B$ , and the creation is undone and then redone by  $A$ , the character should still not be visible:  $B$ 's delete should take effect as if  $A$  never undid the insertion.

This idea leads to the fact that ‘undo insert’ and ‘delete’ operations need to be kept separate. Additionally, ‘undo insert’ needs to take precedence over deletions that have occurred subsequently.

### Implementation

To achieve the desired functionality, I augmented links in the CRDT with a ‘visible’ tag (denoted  $v$ ). The two new bundles that need to be created are undo-insert and redo-insert, which have the same functional form as undo-delete and redo-delete bundles, and only contain the character identifier to operate upon.

The effect of a undo insert packet on CRDT node  $L$  is to set  $L.v = false$ , while a redo insert packet sets  $L.v = true$ . When the string is queried from the CRDT (i.e. the CRDT *read()* method is called), nodes which have  $v = false$  are ignored and not retrieved.

As only the originator of an insert can undo and redo it, and therefore only they can access the  $v$  flag, conflicting operations can never be produced and there is no need to prove commutativity.

### 3.4.3 Undo and Redo of Deletions

Undoing a delete, as mentioned before, is trickier than an insert as multiple clients can concurrently perform and therefore ‘own’ a delete on their local stacks. How this concurrency is handled determines system behavior. The two possibilities are: implementing the CCI consistency model, and a variant I termed “Immediate Undo”, both presented below.

#### CCI Undo/Redo of Deletions

According to CCI, in the event of an undo delete, the system should behave as if the delete never occurred. So, if users 1 and 2 concurrently delete a character, then 1 undoes its removal, the expected result is that the system behaves as if only the 2's delete happened.

While this sounds sensible, it is uncomfortable to user 1: they undid their deletion and expect the character to return. In effect, this consistency model requires *full consensus* between  $n$  clients that concurrently deleted a character to return it.

**Implementation** While consensus is awkward, it is effectively implemented with counters. The key change is to modify the ‘deleted’ tag of links in the CRDT from boolean to integer.

```

1 export interface MapEntry {
2   c: string,    // character
3   n: string,    // next link
4   d?: number,   // (optional) deleted, nonexistent ⇒ not deleted
5   v?: boolean   // (optional) visible, nonexistent ⇒ visible
6 }

```

Listing 3.6: New Type Signature of a Link in the CRDT

Delete and redo operations increment the  $d$  value in the target link. An undo packet has the effect of decrementing  $d$ .

Now, when the *read()* method is called on the CRDT, it only retrieves links such that

$$\forall \text{Links } l. (\neg \exists l.v \vee l.v = \text{true}) \wedge (\neg \exists l.d \vee l.d = 0)$$

**Proof of Commutativity of Delete, Undo, Redo** We only need to consider commutativity with other operations that may operate on the same data in the CRDT and therefore conflict. Since any given delete, undo, or redo operation  $op$  only affects the specific node  $node$  in the linked list identified by  $op.deleteId$ , the only conflicting operations might be other delete, undo delete, and redo delete operations.

*Proof.* We treat delete and redo delete operations identically, that is having the effect of incrementing  $node.d$ . An undo delete has the effect of decrementing  $node.d$ . Any set of these operations applies a sequence of  $+1$  and  $-1$  to  $node.d$ . By the commutativity of addition and subtraction, these arithmetic operations can occur in any order, so the delete, undo and redo operations can occur in any order.  $\square$

**Cost** The cost of this approach is minimal: one extra integer per deleted link in the CRDT.

### Immediate Undo/Redo of Deletions

CCI Undo requires users reach consensus to make deleted characters reappear. It would be preferable for any user to be able to immediately undo a character regardless of other users.

**Semantics** We desire the following semantics:

In the case of conflicting concurrent *make-invisible* operations (such as delete or redo delete) and *make-visible* operations (such as undo delete), the *make-visible* operation will take effect.

This follows from the idea that a user recalling a prior character likely wishes to utilize it, while a user wishing to remove a character does not care about its presence or absence. Note that this effectively inverts the consensus of CCI Undo to require full concurrent agreement to remove a character.

**Implementation** In order to support an immediate undo, we need to be able to compare the “time” at which a deletion, undo, or redo occurred in order to detect concurrency. Vector clocks allow exactly this. When operations arrive, the packet’s vector and the vector stored the relevant CRDT link can be compared and acted upon.

The required modification in the CRDT is that the ‘deleted’ tag  $d$  of a CRDT node  $l$  now represents a pair  $(true/false, vector)$ . The first value is a boolean for whether or not the character is deleted. The second is the result of merging the local vector and the vector of the packet that delivered the operation according to my modified vector clock rules. This executes and returns a pairwise maximum of the two vectors.

As before, delete, undo delete and redo delete bundles contain only the identifier of the node  $l$  to act upon. In non-concurrent cases, i.e. a sequence of causally dependent operations, the last operation in the sequence is applied to  $l$  (intuitively this is the most recent, non-concurrent one).

If a remote delete or redo delete operation is concurrent with the merged vector stored in  $l.d[1]$ , do nothing. This ensures that a local *make-visible* operation retains its effect (e.g. a undo delete). Otherwise, a *make-invisible* operation is already in effect and  $l.d[0]$  is already *true*.

Similarly, if a remote undo delete operation is concurrent with the vector stored in  $l.d[1]$ , always set  $l.d$  to *false*, so that the *make-visible* operation takes effect.

In any case, the vector stored at  $l.d[1]$  is updated to the current client vector clock.

When the CRDT *read()* method is called, we only retrieve nodes  $l$  in the CRDT that satisfy

$$(\neg \exists l.v \vee l.v = true) \wedge (\neg \exists l.d \vee \neg l.d[0])$$

The proof of commutativity can be found in Appendix B.1.

**Cost** Storing a vector with every character that at one point was deleted can become quite expensive: Vector clocks have size  $\Theta(m * \log(k))$ , where  $m$  is the number of clients in the network and  $k$  is the decimal length of the longest sequence number in the vector.

One advantage of this approach is that it enables a sort of distributed garbage collection. Every client is known in the network as it cannot become active until requesting a CRDT from a neighbor (§3.1.3). Clients could record the last known vector sent by every client. Then, they can periodically traverse the local CRDT and remove tombstones whose vectors are strictly older than all the last known vectors of the clients, which guarantees all clients have incorporated the deletion. However the interaction with undo capabilities necessitates further thought and study; this approach is not implemented in this project.

**Alternative Approaches** As seen above, the cost of storing a vector with every deleted element is quite high, though this could be mitigated with vector clock garbage collection. However, it might be preferable to implement similar semantics without vectors. I mentioned before that Immediate Undo in a way inverts consensus to require agreement to make a character disappear. This naturally leads to using counters as in the CCI Undo implementation, but in reverse i.e. counting down from the number of clients in the system rather up from 0. The problem with this approach is that it requires knowledge of how many clients are in the system at all times, and on join or leave all other clients must increment or decrement all counters in tombstones. If a client silently leaves the system, it may never be possible to reach consensus to actually undo or redo a change. This is also an issue present in the CCI Undo implementation.

## 3.5 Summary of Implementation

This chapter described the components I created to evaluate two systems capable of distributed, conflict free text editing. The CRDT-based system was built from the ground up, starting with the conflict-free replicated data type itself. I showed how to optimize it and how to extend its capabilities to include two different forms of undo and redo. To communicate between clients I designed and built a simulated network that guarantees causal delivery, but then also introduced vector clocks to relax requirements on the simulated network. Underlying this network is a scheduler driving the execution of events, some of which are seeded by the experimental setup. The comparative system built around ShareJS executes the same set of events, and the resulting logs from both systems are automatically summarized.



# Chapter 4

## Evaluation

### 4.1 Overall Results

The project can be described as successful if the objectives stated in the proposal are met. These are described below along with brief summaries.

**Success Criterion 1:** *Implement a concurrent, distributed text editor based on CRDTs.*

I can qualitatively assert that this first goal has been completed, complemented by the data gathered and analyzed below. §3.1 outlines the components needed and steps taken to create a distributed text editor based on CRDTs with an underlying simulated network. Though the primary use of the editor is to gather data via programmatically predefined experiments, it can also be used interactively via the user interface. This feature is counted towards fulfilling success criterion 1. ✓

**Success Criterion 2:** *Pass correctness tests for this CRDT.*

The key property of concurrent, distributed text editing is eventual convergence of the data. To this end, the second success criterion aims to ensure that the implementation provides the same guarantees as the theory. Unit testing of the CRDT helped find and eliminate bugs that broke these guarantees. The successful testing is described more in the following section. ✓

**Success Criterion 3:** *Obtain and compare quantitative results from ShareJS and the CRDT-based systems*

This criterion was intentionally vague in the proposal to allow maximum flexibility during analysis. The bulk of this chapter will focus on the results obtained by experimenting on both the CRDT and ShareJS based systems. ✓

## 4.2 Testing the CRDT

The testing of the core CRDT used a *black-box testing* approach [22]. Unit tests were designed for the interface provided by the CRDT without need for knowledge of the CRDT internals. This implies that the CRDT itself could be implemented using a linked list, a linked list within a hash table, or in fact be a completely different CRDT. Insert and delete operations were applied to the CRDT in different orders, and every order needed to produce the same resulting string from `read()`.

The Typescript unit testing library `tsUnit`<sup>1</sup> provided the framework used in this process. The testing output is shown below in Figure 4.1.

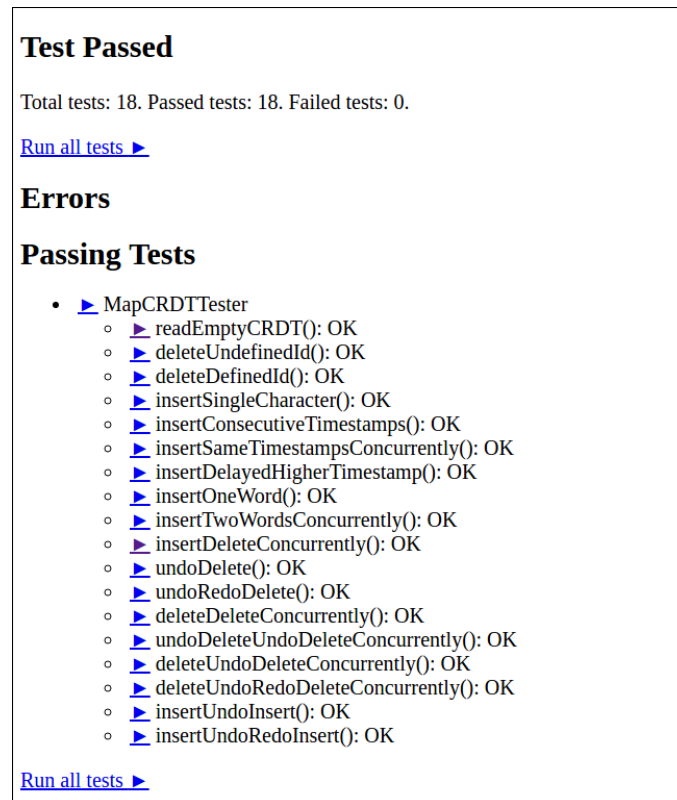


Figure 4.1: Results of various unit tests for CRDT. The undo and redo tests depicted here are for the Immediate Undo variant.

## 4.3 Quantitative Analysis

This section aims to empirically test both ShareJS and the underlying operation transformations, and my CRDT-based system. As experiments were run for data collection rather than testing and debugging, the graphical components were disabled: updating the interface is an expensive operation on both systems and impacts results significantly.

<sup>1</sup><https://github.com/Steve-Fenton/tsUnit>



### 4.3.1 Memory

Data collected for the following sections was done on the Immediate Undo variant (§3.4.3) of the CRDT-based system. I averaged results from both star and fully connected networks after determining topology has little effect on memory consumption. The values presented here are the difference between experiment termination and initialization, to only measure the additional memory consumed.

#### Sanity Check

As noted in §2.2.2, I expected both the ShareJS server and clients to suffer from having to save past operations to transform concurrent modifications against. I created a series of experiments which insert and remove the same character a number of times. This ensures that the visible document size is not growing: the only memory growth should be storing past operations.

For comparison, the same experiments were evaluated on the CRDT system in two ways: firstly using standard insert/delete operations, and secondly as undo/redo operations. The key is that these both have the same effect of inserting and removing characters from the visible document text. However, using insertion and deletion generates a uniquely tagged character each time, so the memory use should increase though the visible document size does not. Doing effectively the same thing with undo and redo operations should not grow internal structures. This test is very useful in establishing the presence of memory leaks and as a general sanity check.

Figure 4.2 confirms these expectations: ShareJS clients and server increase their memory usage as they buffer past operations, though at the end of the experiments the document size is zero characters long. The insert/delete CRDT experiments grow approximately linearly as predicted, whereas using undo/redo causes almost no growth.

This exposes a clear advantage for using CRDT undo/redo versus ShareJS's approach to undo/redo, which generates new operations to undo a prior modification i.e. a delete to undo insertion and an insert to undo a deletion. On the other hand, ShareJS consumes less memory with a static document size and growing number of operations.

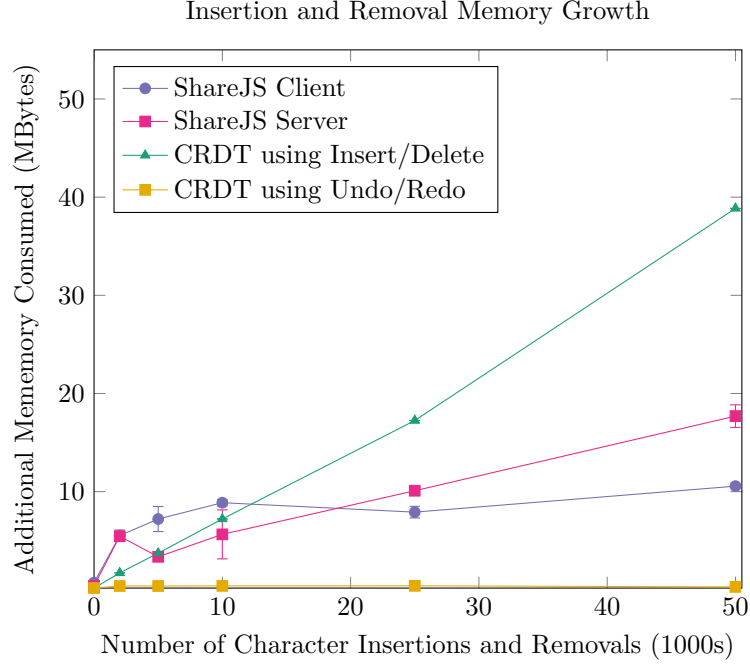
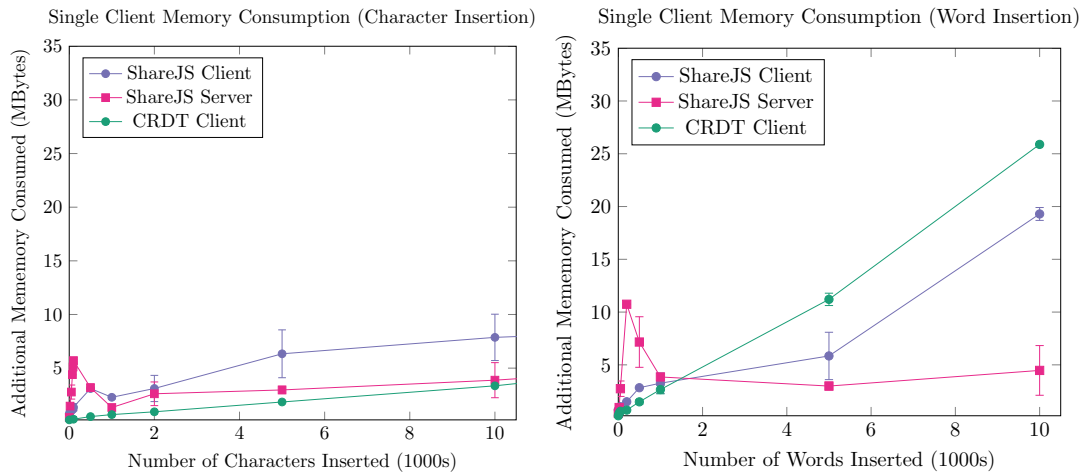


Figure 4.2: A plot comparing the performance of the ShareJS client and server, a CRDT-based client performing insertions and deletions, and a CRDT client using undo and redo operations. The error bars indicate one standard deviation from the mean i.e the variation between experiment trials. Deviations in the CRDT data is generally too small to be visible.

## Single Editor

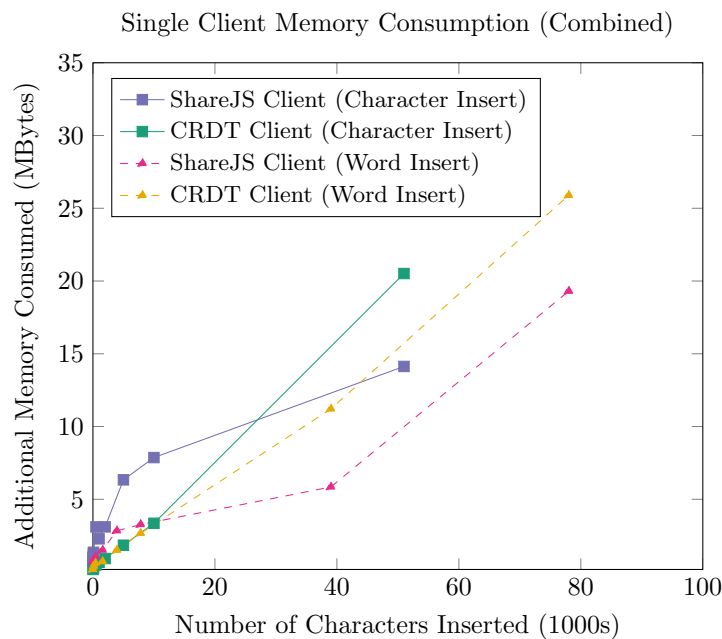
In addition to storing past operations, ShareJS components also need to store the document as it grows. This series of experiment explores combined growth with increasing numbers of insertions of single characters. While ShareJS should scale linearly, a CRDT with  $n$  characters must store  $n$  links of constant size (ignoring deletion requirements), each identified by an identifier of decimal length  $\log_{10}(n)$ , so one might expect a  $\Theta(n \log(n))$  space complexity. These experiments were designed using only a single client, inserting only one character at a time. Results are shown in 4.3a.

To compare a different workload, another series of experiments was performed inserting words of average length approximately 8 characters instead of single letters(4.3b). Both scenarios are plotted on the same axes in 4.3c, with the word insertions rescaled to fit the axis of 4.3a. The data shows that eventually, as the document size becomes very large, ShareJS achieves lower memory consumption. Where this threshold occurs depends on the nature of interactions with the data structures: ShareJS grows less slowly when operations are grouped into fewer, larger packets, reflecting its need to buffer fewer objects in its stack of past operations. On the other hand, my CRDT grows at the same rate no matter what the behavior is (the divergence from 10000 insertions onwards in 4.3c is likely overhead associated with a character by character operations rather than words).



(a) A plot of component behavior as characters are inserted one at a time into the documents.

(b) A plot of component behavior as characters are inserted one word at a time. In this scenario, the CRDT overtakes the ShareJS client after approximately 1300 word insertions, or about  $1300 * 7.8 = 10140$  characters.



(c) A simplified plot including only the ShareJS and CRDT client consumptions on the same axis as 4.3a. The values from 4.3b have been rescaled by the average inserted word length of 7.8 characters.

Figure 4.3: Plots comparing the behavior of ShareJS and CRDT-based systems under different behaviors.

To do a high level estimate of which scenario is more likely, I will assume that a person can type at 60 words per minute, words are of average length 5 characters [3] and worldwide latencies average 100ms [14] [10]. This leads to about 300 characters per minute, or

one character every 200ms. Thus, the character by character streaming scenario that is detrimental to operational transformations is more likely.

### Multiple Editors: Scaling Replication

Figure 4.4 examines how both systems scale with the number of connected clients. In all cases from 1 to 100 connected clients, the CRDT-based distributed editor outperformed the ShareJS based one in terms of overall memory consumption. In the cases of fewer clients, this was by several multiples. Though the CRDT clients appear to exhibit higher aggregate growth as the number of peers increases, part of this can be credited toward overheads in the network simulation.

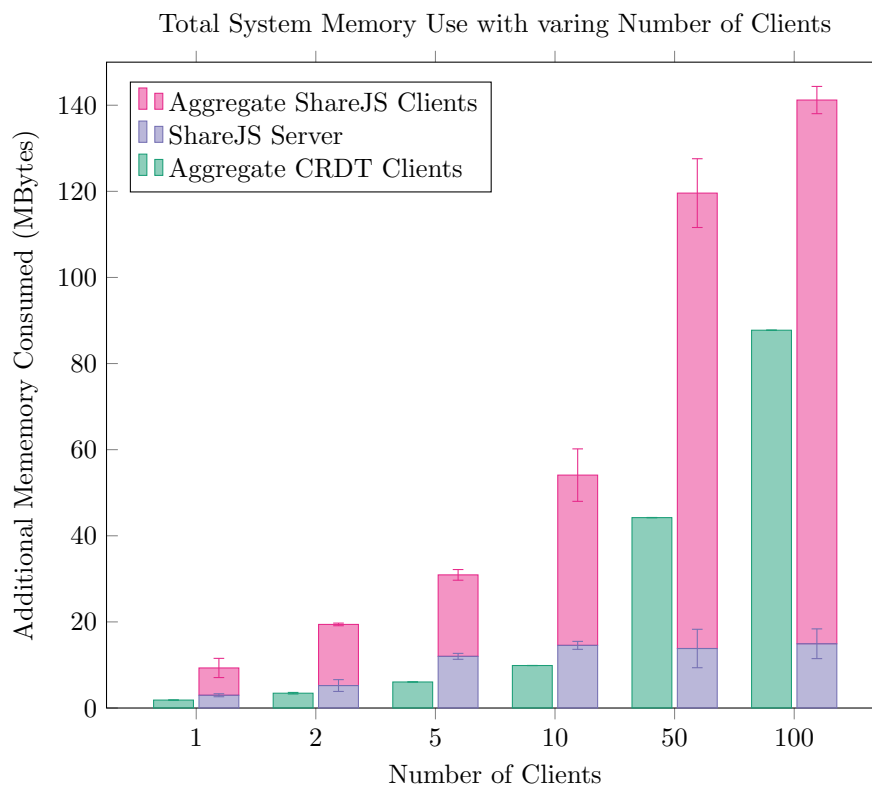


Figure 4.4: Combined server and client cost for the ShareJS system next to the CRDT-based system. Error bars represent one standard deviation. Experiments inserted a total of 5000 characters split across the clients. Error bars in the CRDT are generally too small to be visible.

### CRDT System Variants

Though the prior data sets were collected with arguably the most useful variation presented in §3.4.3 (Immediate Undo) in some situations undo functionality may not be necessary, or CCI consistency may be desired. Thus a comparison between the versions developed is worthwhile: Without undo, CCI Undo, and Immediate Undo; ShareJS is tested as well. This series of experiments tests the systems with varying amounts of

delete operations, but fixed number of clients (5), insertions (100 words per client) and fixed word length (5 characters), 2500 character insertions in total.

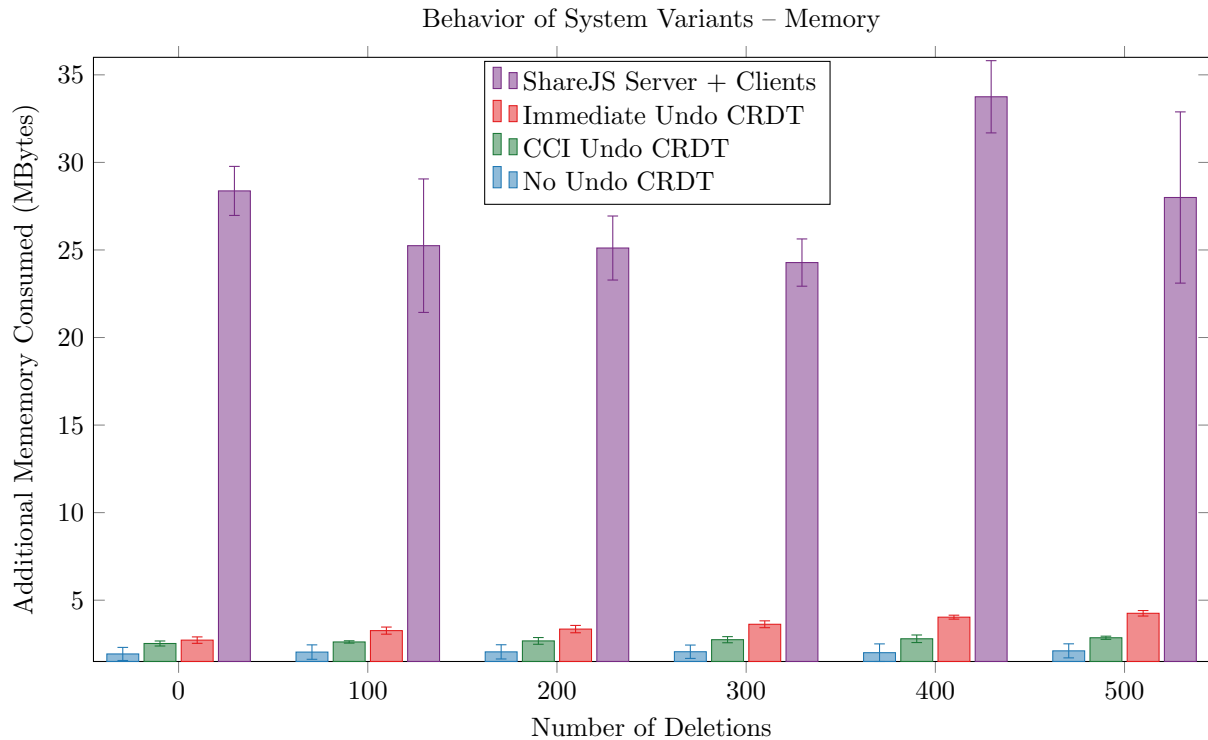


Figure 4.5: A plot of the different system variations developed, along with the ShareJS system with a fixed document size and varying number of deletions. Executed with 5 clients, 100 words inserted per client, each word being 5 characters long.

Figure 4.5 shows that not implementing undo provides consistent performance as hardly any data is added into the CRDT, whereas the Immediate Undo variant grows consistently with the number of characters deleted. CCI Undo sits comfortably between the other variants. ShareJS behaves rather unpredictably, but consumes many times more memory than any of the CRDT-based systems, but would likely perform more comparably with a higher number of insertions.

## Summary

This subsection explored the behavior of both the ShareJS and CRDT-based systems across a variety of settings. Several key ideas were discovered. Firstly, my CRDT offers significantly better memory consumption for small and medium sized documents, though with large documents and chunked operations tends to perform worse. Secondly, OT, at least in the form used by ShareJS, performs worse when operations are small and frequent, whereas CRDTs’ memory consumption is agnostic to the size of operations. Both systems scale relatively similarly as replication increases, though the CRDT generally offered better performance. Lastly, enabling undo functionality has significant overhead, especially when offering ‘Immediate Undo’ semantics, but making effective use of it, per-

haps by replacing as many operations with equivalent undo/redo events as possible, can have considerable savings.

### 4.3.2 Network

#### Latencies and Batching

Figures 4.6 and 4.7 examine the theoretical speed at which operations can be streamed between ShareJS clients, expecting locally looped back transmissions to be the fastest communication possible. Interestingly, a strange feature of these approximate distributions is the 45ms return time peak in Figure 4.7. I traced this into the implementation of the BrowserChannel<sup>2</sup> connection module used by the ShareJS client – messages actually return within a few milliseconds, but are only made available to the client after an additional delay of about 30-40ms. With a trip time via the server to any other client of at most 50ms, the character streaming limit is about 1200 insertions or deletions before they begin being batched into larger changes.

In comparison my system is more flexible: it can be adjusted to send one character at a time, or buffer and group operations to best match the needs of the larger system (for example, if sending small packets is expensive in the network, it can be programmed to send fewer, larger packets like ShareJS).

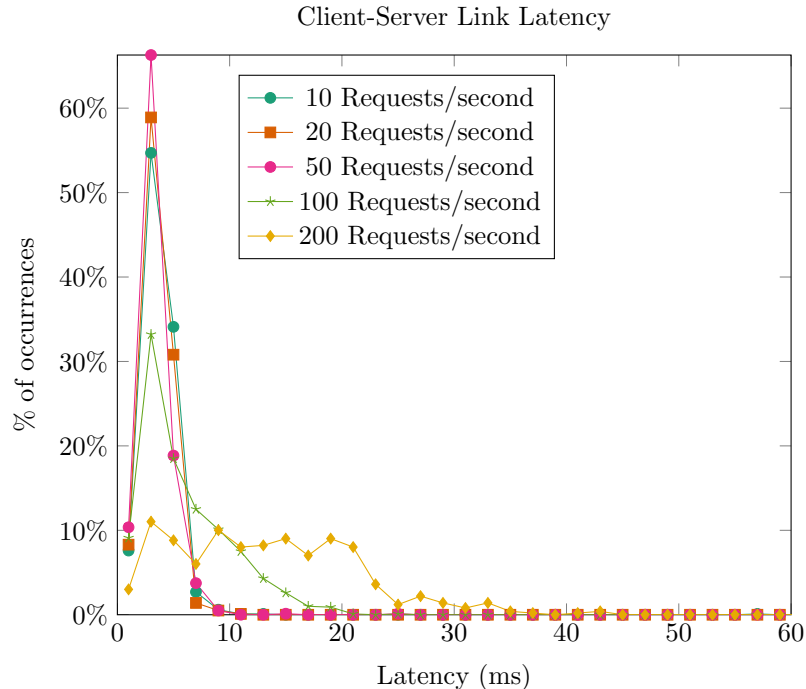


Figure 4.6: A plot of client to server latency versus load. Latencies mostly cluster about 3ms until a higher load is generated.

<sup>2</sup><https://www.npmjs.com/package/browserchannel-middleware>

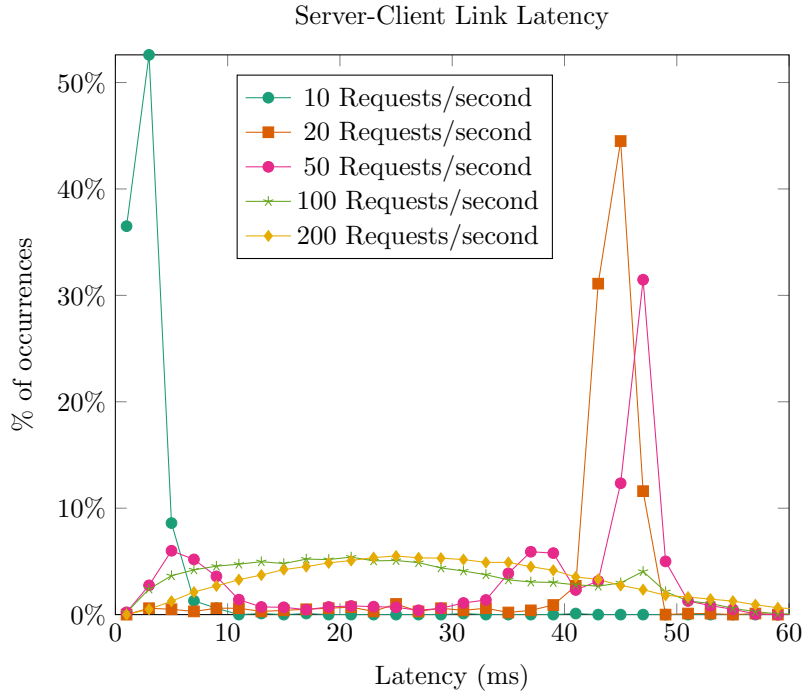


Figure 4.7: A plot of server to client latency versus load. The latencies approach a normal distribution as the load increases.

### Packet Size

The size of packets in both systems is almost fully determined by a few parameters. For this section I will assume that operations are not batched, though they may insert or delete multiple contiguous sequences. As shown above and discussed in §4.3.1, in both this testing environment and real world deployments most packets will contain small operations, so this assumption is reasonable.

Table 4.1: Tabulating the size of ShareJS’s insert and delete packets, measured in number of characters required to represent JSON packet as a string.

	ShareJS Insert	ShareJS Delete
JSON Overhead (chars)	25	25
Length of string to insert $n$	$n$	N/A
Number of characters to delete $n$	N/A	$\lfloor \log_{10}(n) \rfloor + 1$
Position $p$ , bounded by document size $n_{chars}$	$\lfloor \log_{10}(n_{chars}) \rfloor + 1$	$\lfloor \log_{10}(n_{chars}) \rfloor + 1$
Document version $v$ , bounded by $n_{chars}$	$\lfloor \log_{10}(n_{chars}) \rfloor + 1$	$\lfloor \log_{10}(n_{chars}) \rfloor + 1$
Total	$27 + n + 2(\lfloor \log_{10}(n_{chars}) \rfloor)$	$28 + \lfloor \log_{10}(n) \rfloor + 2(\lfloor \log_{10}(n_{chars}) \rfloor)$

Table 4.2: Tabulating the size of the CRDT-based system’s insert packets. Delete packets are identical except they do not contain characters to insert and the ID of the character to insert after. Measured in number of characters required to represent JSON packet as a string.

	Cost (upper bound)
JSON Overhead (chars)	46
Client ID	$\lfloor \log_{10}(n_{clients}) \rfloor + 1$
Vector Clock	$1 + n_{clients} * (\lfloor \log_{10}(n_{operations}) \rfloor + 5)$
Type Disambiguator	1
Characters to Insert	$n$
ID of First New Character	$\lfloor \log_{10}(n_{chars}) \rfloor + \lfloor \log_{10}(n_{clients}) \rfloor + 1$
ID to Insert After	$\lfloor \log_{10}(n_{chars}) \rfloor + \lfloor \log_{10}(n_{clients}) \rfloor + 1$
Total	$51 + \mathbf{n} + \mathbf{5n_{clients}} + \mathbf{n_{clients} * \lfloor \log_{10}(n_{operations}) \rfloor} + 2 * (\lfloor \log_{10}(n_{chars}) \rfloor + \lfloor \log_{10}(n_{clients}) \rfloor)$

The expressions in Tables 4.1 and 4.2 are rather complicated but only need to convey a few ideas: Firstly, packets for insertion scale linearly with the number of characters to add (bolded). Both have significant JSON overheads, though the complexity of the CRDT packets means its overheads are even higher. The vector clock component of the CRDT insert (and delete, not shown here) contributes another  $\Theta(n \log(m))$  cost. This reflects the difficulty of routing and enforcing causality in distributed, flexible topology systems. Lastly, the primary components of these expressions that are likely to grow very large are  $n_{chars}$ , the number of characters in the document, the version  $v$  in ShareJS, and  $n_{operations}$ . However, all of these are present only as  $\log$  terms, so even large values lead to manageable packet sizes.

Table 4.3 shows some typical packet sizes taken from experiments run previously. We see that ShareJS is invariant to the number of clients in the network, whereas the use of vector clocks again penalizes the CRDT-based system. Note that ShareJS packets are shown in two variants: with extraneous meta data stripped out, and with it retained. The meta data allows extra functionality not relevant to this project so should be ignored.


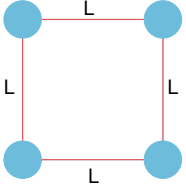
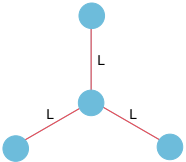
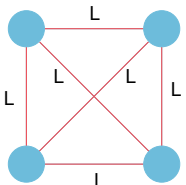


Table 4.3: Sample Real Packet Sizes, measured in number of characters required to represent JSON packet as a string.

Operation	2 Clients	5 Clients
ShareJS Insert 5 chars, 30 char document, without metadata	34	34
ShareJS Insert 5 chars, 30 char document, with metadata	74	74
ShareJS Delete 1 char, 30 char document, without metadata	30	30
ShareJS Delete 1 char, 30 char document, with metadata	74	74
CRDT Insert 5 chars, 30 char document	70	<b>88</b>
CRDT Delete 1 char, 30 char document	54	<b>71</b>

### Packet Quantity

Table 4.4: Table of topology, the number of packets sent per operation, and the time taken until all clients receive the operation the first time. This assumes a constant latency  $L$  on each link. The diagrams in the leftmost column demonstrate topologies with  $n_{peers} = 4$ .

Topology	Total Packets Sent	Time to receipt on all Peers
Linear 	$n_{peers} - 1$	$(n_{peers} - 1)L$
n-gon 	$n_{peers} + 1$	$\lfloor n_{peers}/2 \rfloor L$
Star 	$n_{peers} - 1$	$2L$
Fully Connected 	$n_{peers} * (n_{peers} - 1)$	$L$

Routing in a flexible P2P network is more difficult than in a client-server architecture, which is why my system uses a simple broadcast to disseminate packets. However, this becomes more inefficient with increasing connectivity, illustrated in Table 4.4: the number of packets sent approaches a quadratic cost in the number of peers. Connectivity is one major advantage P2P networks have over traditional ones, as it can lower latency, but requires a more sophisticated routing protocol to handle efficiently.

Using a star topology, ShareJS and the CRDT-based system send the same number of packets into the network. However, payloads are usually larger for the CRDT. Additionally, the *average* time for the clients to receive a modification is lower than in ShareJS: the time to the center node is always  $L$  rather than  $2L$  which brings down the average time for change propagation. However, it would be useful to take advantage of higher connectivity, perhaps using a newer protocol such as Spray [20].

### CRDT System Variants

As before, in some situations it may not be necessary to use vector clocks to ensure causal delivery. In those cases the basic implementation would work well, and undo functionality could be implemented as CCI Undo (as Immediate Undo must have vector clocks). Figure 4.8 demonstrates the cost of using vector clocks once again: packet sizes are double the size versus the basic implementation. ShareJS performs better than the CRDT variants.

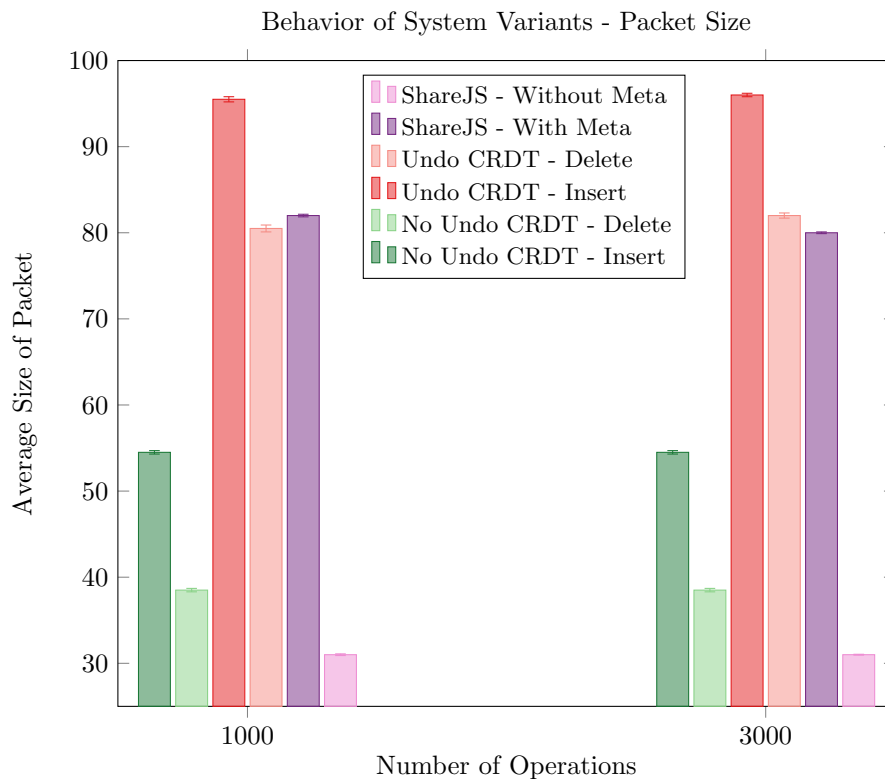


Figure 4.8: Cost of insert and delete packets with different system configurations. Both CCI and Immediate Undo variants use identical packets and are shown here as ‘Undo CRDT’. ShareJS has very similar insert/delete packet sizes; these are shown together.

## Summary

This section showed that using a CRDT over a P2P network offers flexibility in terms of operation grouping or streaming that ShareJS does not. In addition, on equivalent topologies they send the same quantity of packets, but my system sends packets that are between 1.5 and 3.5 times as large, reflecting the use of P2P networks. Relatedly, using a P2P network can halve the latency to disseminate operations to all peers, but this either comes at the cost of a more complex network protocol or massively increased numbers of packets. Overall, ShareJS generally requires less network capacity and offers reasonable latencies to all clients.

### 4.3.3 CPU Time

Knowing the relative simplicity of CRDT algorithms versus those of OT hints that processing time may be the real benefit of using CRDTs. Especially bad are scenarios where OT has to do costly transformations of incoming operations against a long history of concurrent ones; this issue does not arise with CRDTs.

Unfortunately, web browsers are nonideal platforms for examining computation time (cannot actually measure CPU time). Additionally, nondeterministic latency between ShareJS clients and server could make results unreliable. To handle these issues, I chose to generate very large concurrent operations for both systems and use millisecond-precision timers to measure the time taken from operation arrival from the network until complete integration. Large workloads disguise the use of low precision timers and hide effects of operating system level scheduling. In these experiments, one client is a simple listener and receives operations generated concurrently on other nodes. The times indicated in Figure 4.9 are the sum of processing times on generator nodes, server relay time (if applicable), and integration time on the listener.

As shown in the graph, ShareJS spends much longer generating and integrating changes than the CRDT-based system does. The CRDT operates between 3 and 38 times faster than ShareJS. This data provides strong evidence that CRDTs fare much better under highly concurrent loads. This is in line with what has been found by other authors [1].

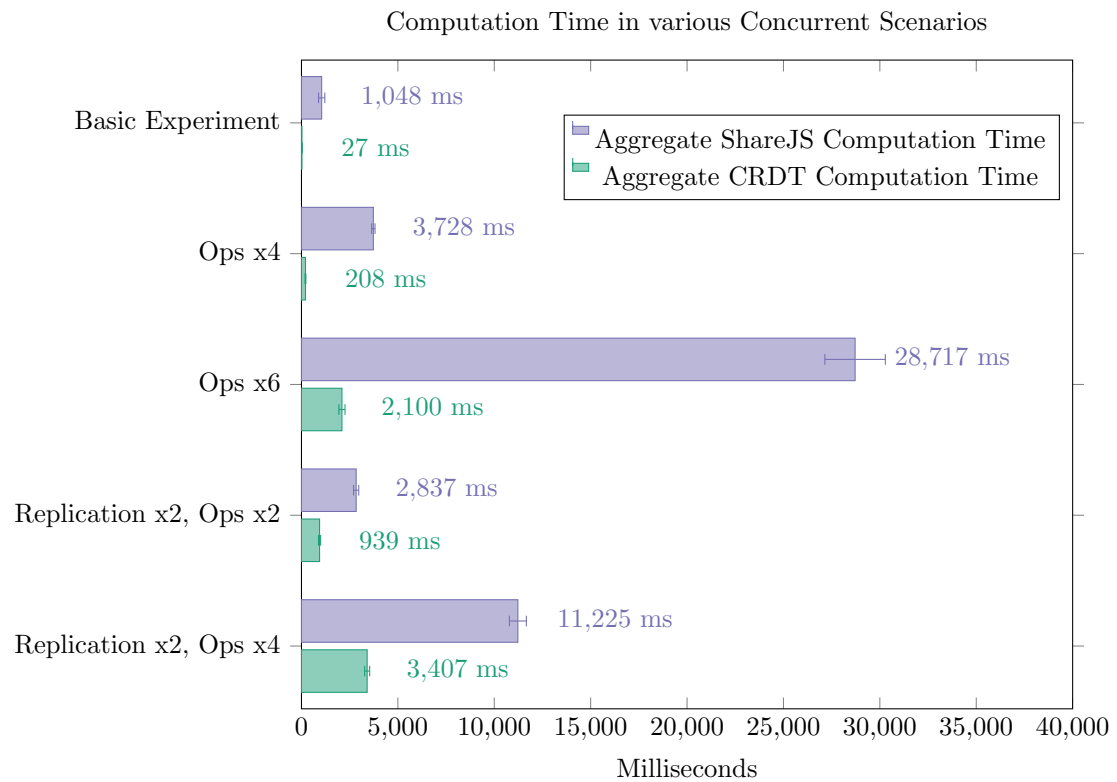


Figure 4.9: Comparison between the time taken for both systems to integrate large concurrent edits.

# Chapter 5

## Conclusion

Though realtime collaboration is more feasible than ever, aspects of currently deployable technologies still need improvement. Operational transformations (OT), which power popular services such as Google Docs, are magnitudes slower to integrate concurrent operations and settle into a converged state than alternative approaches driven by convergent replicated data types (CRDTs). CRDTs are also theoretically simpler and therefore easier to implement without errors. The data- rather than operation-centric approach to eventual consistency utilized by CRDTs allows exploitation of P2P networks, enabling more robust and independent systems to be developed, though this comes at the cost of network traffic and routing complexity. My findings, the apparent stagnation in development of new OT algorithms, and the continuing potential for optimization of CRDTs suggest that future systems will opt for CRDTs over OT for eventual consistency.

### 5.1 Achievements

I succeeded in meeting all my initial project goals: the final CRDT passed a set of correctness tests I defined, which in turn enabled correct operation of a distributed text editor running on top of a simulated network. To enable comparison, I built a system around the OT-enabled ShareJS capable of executing equivalent experiments as my from-scratch editor. In §4.3, I presented quantitative analysis exploring various dimensions of both systems: memory usage, network behavior and computation time, across different document sizes and levels of replication.

In addition to the core goals, I completed one extension which implements two different undo and redo capabilities for the CRDT, each providing different semantics. My work goes beyond that of the creators of the RGA CRDT and makes it as capable as ShareJS for text editing. Data I gathered indicates that implementing the “Immediate Undo” semantics has a relatively high cost, but also potential for large savings over the OT approach to undo and redo used in ShareJS.

This project has from the outset been designed to facilitate replacement of the network simulation with a real P2P networking stack. Having this overall goal in mind led me to plan my implementation thoroughly and isolate functionality. The result is a pleasingly modular, extensible piece of engineering that I hope to build on in the future. Aside from the software development perspective, I gained much deeper insight into conflict free algorithms and data structures, reinforced much of the theoretical underpinnings learned in coursework with practical implementation, and discovered how much fun it could be to spend days reading scientific papers and doing real research.

## 5.2 Lessons Learned

The task of taking an idea and creating a fully fledged, useful implementation is incredibly satisfying. However, in hindsight, there are a few adjustments I would make to the development process. I would spend more time focusing on the CRDT, rather than the networking aspect of the project, implementing further optimizations and capabilities. Additionally, getting the network simulation and schedulers working correctly may well have taken just as long as implementing a real network stack and not been worthwhile. Lastly, I underestimated the amount of time it would take to fix memory leaks and complete accurate, comparable evaluations of the systems, and would start these steps earlier.

## 5.3 Limitations and Future Work

My CRDT is fundamentally a replicated linked list designed specifically for text editing. As such it does not extend particularly well to other sorts of data that might want to be shared, such as JSON. Within the domain of text editing, additional functionality such as copy-paste combined with undo would be difficult, if not impossible, to implement. Only recently have CRDTs been developed which are capable of these block actions [35].

However, there remains potential for further work in my system, mostly to do with optimizations.

- **CRDT Optimizations:** I implemented word insertion at the network level. That is, rather than sending one character with an identifier per packet, sending one word with an identifier only for the first character. This could be extended to the CRDT itself: storing contiguous words compactly and splitting them on demand could reduce the space complexity of the CRDT by a large multiplier. Other ideas include tombstone garbage collection using vector clocks.
- **Probabilistic Broadcast:** As discussed previously, flooding is a relatively inefficient delivery mechanism in a P2P network. Instead, a probabilistic broadcast algorithm such as lpbcast combined with an anti-entropy protocol ([9]) could be used, reducing the network load while taking advantage of the connectedness of a

P2P network. Alternatively a more modern approach such as Spray [20] could be used.

- **Vector Clocks:** Using full vector clocks overly restricts concurrency as all potentially dependent operations are stalled, rather than only those with real causal requirements. This could be made more efficient by listing dependent operations, again resulting in a form of vector. However, more efficient representations have been developed such as dotted version vectors [24].





# Bibliography

- [1] Mehdi Ahmed-Nacer et al. “Evaluating crdts for real-time document editing”. In: *Proceedings of the 11th ACM symposium on Document engineering*. ACM. 2011, pp. 103–112.
- [2] Hagit Attiya et al. “Specification and complexity of collaborative text editing”. In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. ACM. 2016, pp. 259–268.
- [3] Vladimir V Bochkarev, Anna V Shevlyakova and Valery Solovyev. “Average word length dynamics as indicator of cultural changes in society”. In: *CoRR* (2012).
- [4] Quang-Vinh Dang and Claudia-Lavinia Ignat. “Performance of real-time collaborative editors at large scale: User perspective”. In: *IFIP Networking Conference (IFIP Networking) and Workshops, 2016*. IEEE. 2016, pp. 548–553.
- [5] Stephen E. Deering and Robert M. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. Accessed: 2017-04-01. Dec. 1998. URL: <http://www.rfc-editor.org/rfc/rfc2460.txt>.
- [6] Dr. Andrew Moore. *Computer Networking*. Accessed: 2017-04-01. 2016. URL: <http://www.cl.cam.ac.uk/teaching/1516/CompNet/>.
- [7] Dr. R.J. Gibbens. *Computer Systems Modeling*. Accessed: 2017-04-01. 2017. URL: <http://www.cl.cam.ac.uk/teaching/1617/CompSysMod/>.
- [8] C A Ellis and S J Gibbs. “Concurrency Control in Groupware Systems”. In: (1989).
- [9] P Th Eugster et al. “Lightweight probabilistic broadcast”. In: *ACM Transactions on Computer Systems (TOCS)* 21.4 (2003), pp. 341–374.
- [10] FCC. *Measuring Broadband America*. Accessed: 2017-04-25. URL: <https://www.fcc.gov/general/measuring-broadband-america>.
- [11] Colin J Fidge. “Timestamps in message-passing systems that preserve the partial ordering”. In: (1987).
- [12] Joseph Gentle. *ShareJS v0.6.3*. <https://github.com/josephg/ShareJS/tree/0.6>. Accessed: 2016-10-09. 2013.
- [13] Seth Gilbert and Nancy Lynch. “Brewer ’ s Conjecture and the Feasibility of Consistent , Available , Partition-Tolerant Web Services”. In: (2005), pp. 51–59.
- [14] Ilya Grigorik. *Internet Latencies*. Accessed: 2017-04-25. URL: <https://www.igvita.com/2012/07/19/latency-the-new-web-performance-bottleneck/>.
- [15] Google Inc. *Protocol Buffers*. Accessed: 2017-04-20. URL: <https://developers.google.com/protocol-buffers/>.

- [16] Santosh Kumawat and Ajay Khunteta. “Analysis of Operational Transformation Algorithms”. In: Springer. 2016, pp. 9–20.
- [17] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Communications of the ACM* 21.7 (1978), pp. 558–565.
- [18] Brice Nédelec et al. “Concurrency Effects Over Variable-size Identifiers in Distributed Collaborative Editing”. In: *Proceedings of the International workshop on Document Changes: Modeling, Detection, Storage and Visualization, Florence, Italy, September 10, 2013*. Vol. 1008. 2013, pp. –7.
- [19] Brice Nédelec et al. “LSEQ: an adaptive structure for sequences in distributed collaborative editing”. In: *Proceedings of the 2013 ACM symposium on Document engineering*. ACM. 2013, pp. 37–46.
- [20] Brice Nédelec et al. “Spray: an Adaptive Random Peer Sampling Protocol”. PhD thesis. LINA-University of Nantes; INRIA Rennes-Bretagne Atlantique, 2015.
- [21] Gérald Oster et al. “Proving correctness of transformation functions in collaborative editing systems”. PhD thesis. INRIA, 2005.
- [22] Ron Patton. *Software Testing*. Indianapolis, IN, USA: Sams, 2005.
- [23] Nuno Preguica et al. “A commutative replicated data type for cooperative editing”. In: *Distributed Computing Systems, 2009. ICDCS’09. 29th IEEE International Conference on*. IEEE. 2009, pp. 395–403.
- [24] Nuno Preguiça et al. “Dotted version vectors: Logical clocks for optimistic replication”. In: (2010).
- [25] Hyun-Gul Roh et al. “Replicated abstract data types: Building blocks for collaborative applications”. In: *Journal of Parallel and Distributed Computing* 71.3 (2011), pp. 354–368.
- [26] Marc Shapiro and Nuno M. Preguiça. “Designing a commutative replicated data type”. In: *CoRR* abs/0710.1784 (2007).
- [27] Marc Shapiro et al. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. INRIA, 2011, p. 50.
- [28] B E N Shneiderman. “The future of interactive systems and the emergence of direct manipulation”. In: *Behaviour & Information Technology* 1.3 (1982), pp. 237–256.
- [29] Chengzheng Sun and Clarence Ellis. “Operational transformation in real-time group editors: issues, algorithms, and achievements”. In: ACM. 1998, pp. 59–68.
- [30] Mikito Takada. *Distributed Systems: for fun and profit*. 2013, pp. 1–62.
- [31] *The Mother of All Demos, Reel 3*. [https://archive.org/details/XD300-25\\_68HighlightsAResearchCntAugHumanIntellect&start=286](https://archive.org/details/XD300-25_68HighlightsAResearchCntAugHumanIntellect&start=286). Accessed: 2017-04-01.
- [32] Dr. Robert N. M. Watson. *Concurrent and Distributed Systems*. Accessed: 2017-04-01. 2016. URL: <https://www.cl.cam.ac.uk/teaching/1617/ConcDisSys/>.
- [33] Stephane Weiss, Pascal Urso and Pascal Molli. “Logoot-undo: Distributed collaborative editing system on p2p networks”. In: *IEEE Transactions on Parallel and Distributed Systems* 21.8 (2010), pp. 1162–1174.
- [34] Stéphane Weiss, Pascal Urso and Pascal Molli. *Logoot: a P2P collaborative editing system*. Research Report RR-6713. INRIA, 2008, p. 13.

- [35] Weihai Yu, Luc André and Claudia-Lavinia Ignat. “A CRDT supporting selective undo for collaborative text editing”. In: *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer. 2015, pp. 193–206.
- [36] Peter Zeller, Annette Bieniusa and Arnd Poetzsch-Heffter. “Formal specification and verification of CRDTs”. In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer. 2014, pp. 33–48.



# Appendix A

## Vector Clocks

### A.1 Formal Definition

As outlined in [11], the aim of vector clocks is to implement the  $\rightarrow$  relation such that  $a \rightarrow b$  iff  $a$  can causally affect  $b$ . It is possible to have neither  $a \rightarrow b$  nor  $b \rightarrow a$  indicating concurrent events.

A vector clock  $VC$  is an array of integer timestamps  $[c_1, c_2 \dots c_n]$ .  $c_p$  represents the last known clock value  $c$  of process  $p$ ,

There are 5 rules associated with traditional vector clocks.

1. All values are initially zero.
2. The local clock value is incremented at least once for each atomic event.
3. The current value of the entire vector is sent with every outgoing message.
4. Upon receipt of a vector, set the local vector to be the maximum of corresponding values in the received and local vectors. The local value is incremented by one.
5. No value is ever decremented.

The desired partial ordering can then be defined

$$VC_a <_v VC_b \leftrightarrow \forall i[VC_a[i] \leq VC_b[i]] \wedge \exists j[VC_a[j] < VC_b[j]]$$

### A.2 Modified Vector Clock

The modified rule used in §3.1.3 changed rule four from the previous section. For completeness, the full modified rules used are listed below.

1. All values are initially zero.

2. The local clock value is incremented with every message sent by a replica.
3. The current value of the entire vector is sent with every outgoing message.
4. Upon receipt of a vector, set the local vector to be the maximum of corresponding values in the received and local vectors.
5. No value is ever decremented.

# Appendix B

## Convergence of Immediate Undo Variant

### B.1 Proof of Commutativity

Section §3.4.3 presented an alternative method of implementing undo and redo functionality for the CRDT. To prove convergence, only commutativity needs to be guaranteed as discussed in §2.2.3. The commutativity of the undo and redo operations are presented below.

We only need to consider commutativity with other operations that may operate on the same data in the CRDT and therefore conflict. Since any given delete, undo, or redo operation  $op$  only affects the specific node  $node$  in the linked list identified by  $op.deleteId$ , the only conflicting operations might be other delete, undo delete, and redo delete operations.

*Proof.* Treat delete and redo delete operations identically. Any non-concurrent modifications are not considered as the latest causally dependent operation is defined to take effect.

Take a system which begins in quiescence, submits concurrent operations and returns to quiescence. There are three possible conflicting cases: delete-delete, delete and delete-undo, and undo-undo.

In the delete-delete scenario, the operations are idempotent: both set  $node.d[0]$  to *true* and  $node.d[1]$  to the merged local and incoming vectors of the two operations. Thus on any client, the target node is no longer visible. The undo-undo case is exactly analogous except  $node.d[0] = false$ .

In the delete, delete-undo case (one client deletes while another deletes then undoes immediately), any client receiving a single delete  $op_{d_1}$  first will set  $node.d[0] = true$  and record the merged vector in  $node.d[1]$ . On delivery of the concurrent delete  $op_{d_2}$ , only the vector at  $node.d[1]$  is updated as the node is already marked as deleted. Then, undo

$op_{undo_2}$ , since it is a *make-visible* operation, and determined to be concurrent with the merged vector stored at  $node.d[1]$ , takes effect and  $node.d[0]$  is set to *false*. If  $op_{d_2}$  were delivered before  $op_{d_1}$  the same would occur. Lastly, if the first arrivals are  $op_{d_2}$  followed by  $undo_{d_2}$ , the undo takes effect and  $node.d[0] = false$  and  $node.d[1]$  is set to the merged vector. On arrival of  $op_{d_1}$ , since it is concurrent with  $node.d[1]$  and a *make-invisible* operation taking effect against a *make-visible* operation, it is ignored. Thus, in any case the clients resolve  $node.d[0] = false$  and the node is visible. The stored vector is the result of merging the vectors of the three operations. This merge will produce identical results in any order as proven in Lemma 1.

□

**Lemma 1.** *A set of vectors  $S = v_1, v_2 \dots v_n$  merged in any order will converge to the same result.*

*The vector merge rule used is Rule 4 from Appendix A.2, which states the merge between vectors  $v_a$  and  $v_b$  is the component-wise maximum of the vectors. That is,  $\forall i. v_{result}^i = \max(v_a^i, v_b^i)$ .*

*The result of merging  $n$  vectors is  $v_{result}^i = \max(v_1^i, \max(v_2^i, \dots \max(v_{n-1}^i, v_n^i) \dots))$ , though the order of nesting is arbitrary. In any ordering component  $i$  is simply the maximum of all possible  $i$  components from the set  $S$ . Thus, the ordering used to merge a set of vectors is irrelevant and produces the same result in any ordering.*



# Appendix C

## Simple Experiment Experimental Setup and Results

### C.1 Experiment Setup

```
{
  // Experiment Name
  "experiment_name": "experiment_1",
  // How many clients and when they join
  "clients": [0,0,0],
  // CRDT-only: Per client values used to compute link latencies
  "network": {
    "0": {"latency": 196.7632081023716},
    "1": {"latency": 220.01040150077455},
    "2": {"latency": 122.31541467483453}
  },
  // CRDT-only: type of scheduler to use
  "execution": "event-driven",
  // Scheduled events
  "events": {
    "0": {
      "insert": {
        "0": {
          "chars": "coliseums",
          "after": 0
        }
      },
      "delete": {}
    },
    "1": {
      "insert": {
        "15": {
          "chars": "tackling",
          "after": 0
        }
      }
    }
  },
  // Events for client 0
  // Insert events for client 0
  // Insert "coliseums" at time 0, index 0
  // No delete events
  // Events for client 1
  // Insert events for client 1
  // Insert "tackling" at time 15, index 0
}
```

```

        "delete": {}                                // No delete events
    },
    "2": {                                           // Events for client 1
        "insert": {                                  // Insert events for client 1
            "30": {                                   // Insert "freshening" at time 30, index 0
                "chars": "freshening",
                "after": 0
            }
        },
        "delete": {}                                // No delete events
    },
    // CRDT-only: Topologies over which to run the same experiment
    "topology": [
        "fully-connected",
        "star"
    ]
}

```

## C.2 Summary of Results of Simple Experiment

---fully-connected, optimized---

```

Total simulation duration: 848.547219206
Optimizations enabled: True
All clients converged to same result: True
Total insert events: 3
Total delete events: 0
Total insert packets sent: 13
Total size of insert packets sent: 1329
Average insert packet size (incl vector clock etc.): 102.230769231
Total delete packets sent: 0
Total size of delete packets sent: 0
Average delete packet size (incl vector clock etc.): 0
Expected number of packets sent - given naive broadcast in a p2p network: 18
Expected number of packets sent - given optimal p2p network with everyone
  joining at start: 6
Latency/wait time per client when requesting CRDT: [0, 416.77360960314616,
  319.07862277720614]
From whom each client request CRDT: [-1, 0, 0]
Length of stringified document/crdt during state replay, on average: 171
pre-experiment: 0
post-topology-init: 149136
post-graph-init: 1114064
post-clients-init: 1125632
post-experiment: -1833488

```

---star, optimized---

```

Total duration: 959.239037182
Optimizations enabled: True
All clients converged to same result: True
Total insert events: 3
Total delete events: 0

```

```

Total insert packets sent: 9
Total size of insert packets sent: 917
Average insert packet size (incl vector clock etc.): 101.888888889
Total delete packets sent: 0
Total size of delete packets sent: 0
Average delete packet size (incl vector clock etc.): 0
Expected number of packets sent - given naiive broadcast in a p2p network: 12
Expected number of packets sent - given optimal p2p network with everyone
  joining at start: 6
Latency/wait time per client when requesting CRDT: [0, 416.77360960314616,
  319.07862277720614]
From whom each client request CRDT: [-1, 0, 0]
Length of stringified document/crdt during state replay, on average: 171
pre-experiment: 0
post-topology-init: 73776
post-graph-init: 1485920
post-clients-init: 1493768
post-experiment: 1192544

```

```

---experiment_1, ot---

```

```

Total simulation duration: 2100.0
Total insert events: 3
Total delete events: 0
Total packets: 18
Total size of packets sent: 1398
Average packet payload size: 77.6666666667
Total size of packets sent, if there were no meta-information: 264
Average packet payload size without meta-information: 14.6666666667
Expected number of packets sent - give optimal client-server network with
  everyone joining at start: 18
Latency/wait time per client when requesting CRDT: [43.0, 18.0, 19.0, 0]
From whom each client request CRDT: [-1, -1, -1, -1]
Length of stringified document/crdt during state replay, on average: 0
pre-experiment: 0
post-clients-create: 2170512
post-clients-init: 716032
post-experiment: 1101280

```



# Appendix D

## Project Proposal

Computer Science Tripos – Part II – Project Proposal

### Conflict Free Document Editing with Different Technologies

J. Send, Trinity Hall

Originator: J. Send

10 October 2016

**Project Supervisor:** S. Kollmann

**Director of Studies:** Prof. S. Moore

**Project Overseers:** Prof. T. Griffin & Prof. P. Lio

## Introduction

### Background

In a world of ever increasing connectivity, collaborative features of applications will take on greater and greater roles. Popular services such as Google Docs offer real-time editing of documents by multiple users, a type of interaction that will move from being a special offering by few applications to a common and expected interface.

The key property that must be implemented to achieve concurrent editing is eventual consistency, meaning that all connected users should end up with the same result after receiving all changes to the document — even if edits conflict [2]. There are two main

technologies that are used to enable concurrent editing of a document (plain text or otherwise). One approach is Operational Transforms (OT), which generally relies on having a central server receive, serialize, transform, and relay edits occurring simultaneously to each client. OT is notoriously difficult to implement correctly as incoming operations have to be transformed against preceding ones on each client, such that the result converges [29]. The server may also be required to make some transformations. Due to this, central server must be able to read all the operations being performed by clients. Thus, unless the server is trusted and secure, OT-based services cannot provide any security or privacy guarantees.

The alternative, newer technology uses Conflict Free Replicated Datatypes (CRDTs). Instead of resolving conflicts and guaranteeing eventual consistency by transforming operations against each other, CRDTs use special datastructures that guarantee that no operations will conflict [23]. There are many types of CRDTs that are tailored for different situations. One example is a simple up-down counter which could be implemented as two locally replicated registers, one for increments and one for decrements, where the current state is their difference[27]. Compared to OT, there is no interdependence between edits (as long as the network protocol can guarantee in-order delivery), which means CRDT-based systems can do away with the server and be implemented using peer to peer (P2P) protocols. This lends itself to security (encryption is now possible between endpoints), and possibly more scalability and efficiency.

This project is first concerned with exploring and developing a P2P CRDT concurrent text editor, and secondly comparing it to the OT-based client/server approach available in the open source library ShareJS. Several extensions are also possible, listed in later sections.

## Resources required

The primary external resource I will need is the Javascript library ShareJS, which is published under the MIT license on GitHub [12].

Additionally, I am developing on my personal computer, a Thinkpad T440s with 8 GiB of RAM, 128 GB of hard drive space, and a low wattage dual core Intel CPU running at 1.60GHz. The primary development environment is Ubuntu Linux, though Windows 10 is also available on the same machine.

Git with Github is used as both a version control system and a cloud backup. Dropbox provides continuous cloud backups as well. Secondary development machines are any of the MCS computers.

## Starting point

I have some knowledge of the open source library ShareJS from a past internship, which I aim to leverage when evaluating and comparing it to my system. My knowledge of

CRDTs and the relevant adding/insert/merge algorithms stems mostly from a high level explanation provided by Martin Kleppmann, along with a diagram. This will be the starting point for my from-scratch implementation of the concurrent text editor.

Since I have no experience writing test cases and performance profiling, nor network simulation, I will have to learn how to do these.

Lastly, I may consult various papers on CRDTs, as well as my supervisor's work in the area, if required.

## Work to be done

### Overview

I plan to implement a simulation of P2P CRDT text editing using Typescript. Following this, my project will focus on comparing an existing OT-based concurrent document editing library (ShareJS) to my implementation, in order to draw conclusions about their relative network and memory efficiency, and scalability. It is highly likely that my system will need some optimization, which can feed back into my evaluation and comparison process. In the case that these phases do not take too long, there are several possible extensions. The first would be to add a networking layer to the simulation - in effect turning the it into a usable library. The second would be researching and implementing 'undo' and 'move' operations, which are relatively open research problems.

### Detailed Project Structure

1. **Core CRDT Development:** Consider and decide CRDT datastructures. Then detail how I expect the insert/delete/merge algorithms to work on paper, followed by implementing these. Lastly, I need to learn frameworks for testing my implementation. The tests for correctness should include hand-crafted unit tests to confirm expected behavior of intermediate execution steps and convergence of results across clients, along with generated test loads to check correct convergence on all clients.
2. **Implement Simulation:** Model having an arbitrary number of clients each running the CRDT algorithms, and simulate networking between these clients. Because this is P2P, it may be worth adding functionality for a variety of network topologies.
3. **Set up ShareJS and Compare:** Set up the ShareJS environment, mirror functionality and setups between two systems as much as possible, and create corresponding performance profiling tests for both systems. These will focus on network efficiency, memory usage, and scalability.

4. **Tune Implementation:** There will likely be opportunity for some optimization, which will feed back into the performance comparisons in the previous step and help evaluate the optimizations themselves.
5. **Extensions:** The first extension is implementing a proper P2P network stack and remove the simulated networking. Next would be researching undo and move operations and perhaps try to implement one or both of these.

## Possible extensions

There are two extensions of varying difficulty:

- (Easier) Replace networking simulation with a P2P networking library. The end result of this extension should be a ready to deploy Typescript (compiled to Javascript) library.
- (Difficult) Research prior work on undo and move functionality using CRDTs. If something suitable is found, implement it. Otherwise, attempt to work toward my own solution.

## Success criteria

These are the main success criteria associated with my project

1. A concurrent text editor based on CRDTs has been implemented.
2. The concurrent text editor passes all correctness tests.
3. Quantitative results comparing ShareJS and the CRDT based system have been obtained and analyzed.

## Timetable

Planned starting date is 16/10/2011.

1. **Michaelmas weeks 2–3** Develop CRDT datastructure and algorithms on paper. Read into P2P networks and simulating them.
2. **Michaelmas weeks 4–5** Lay out project files and implement network simulation with support for different P2P topologies.
3. **Michaelmas weeks 6–8** Implement CRDT datastructures and algorithms, and connect these to network simulation.



4. **Michaelmas vacation** Learn an appropriate testing framework, write and generate unit tests for correctness of implementation. Fix any bugs discovered by the testing process. Set up ShareJS environment. Begin outlining progress report.
5. **Lent weeks 0–1** Complete progress report. Mirror functionality of ShareJS to the setup of my system. Start writing performance benchmarks and scalability tests for both systems.
6. **Lent weeks 2–4** Execute tests and analyze results. Try to explain differences and similarities observed. Tune my implementation and evaluate various optimizations. Begin writing dissertation.
7. **Lent weeks 5–6** Continue writing dissertation and optimizing system. Begin research for extension which implements proper networking stack.
8. **Lent weeks 7–8** Continue writing dissertation. Before terms ends, review and peer-review (including supervisor) incomplete draft. Implement networking extension. Research undo and move operations with CRDTs.
9. **Easter vacation:** Finish dissertation draft. Work on undo and move extensions for system.
10. **Easter term 0–2:** Edit and proof read dissertation. Work on extensions.
11. **Easter term 3:** Proof read and then submit early to concentrate on exams.