Joshua Send

Conflict Free Document Editing with Different Technologies

Computer Science Tripos – Part II

Trinity Hall

22nd April 2017

Proforma

Name:	**Joshua Send**
College:	**Trinity Hall**
Project Title:	**Conflict Free Document Editing with Different Technol**
Examination:	**Computer Science Tripos – Part II, June 2017**
Word Count:	**1587**[1]
Project Originator:	Joshua Send
Supervisor:	Stephan Kollmann

Original Aims of the Project

TODO[2]

Work Completed

TODO

Special Difficulties

TODO

---

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

[2] A normal footnote without the complication of being in a table.

Declaration

I, Joshua Send of Trinity Hall, being a candidate for Part II of the Computer Science Tripos [or the Diploma in Computer Science], 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 TODO [signature]

Date TODO [date]

Contents

List of Figures

Acknowledgements

TODO

Chapter 1

Introduction

Real time interaction between users is becoming an increasingly important feature to 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 use 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 motivated by a demonstration in the Mother of All Demos by Douglas Engelbart in 1968 [17]. From that time, it took several decades for implementations of such editing systems to appear. Early products were released in the 1990's, and the 1989 paper by Gibbs and Ellis [4] marked the beginning of extended research into operational transformations. Due to almost 20 years of research, OT is a relatively developed field and has been applied to products that are commonly used. The most familiar of these is likely to be Google Docs[1], which seems to behave in a predictable and well understood way. One reason one reason for this is that it follows users' expectations for how a concurrent, multi-user document editor should work. Importantly, this includes lock-free editing and independence of a fast connection, no loss of data, and the guarantee that everyone converges to the same document when changes are complete. These are in fact the goals around which OT and CRDTs have developed.

The convergence, or consistency, property above is the hardest to provide – it is easy to create a system where the last writer wins, but data is lost in the process. In a distributed system such as a shared text editor, the CAP theorem tells us we cannot guarantee all three of consistency, availability, and partition-tolerance [7]. However, if we forgo strong consistency guarantees and settle for eventual consistency, we are able provide all three

---

[1]https://docs.google.com

[21]. As we will see, achieving eventual consistency is non-trivial. The two prevailing approaches that enable it, operational transformations and commutative replicated data structures, are discussed in detail the Preparation section.

1.2   Overview

This project aims to examine the trade-offs made when implementing highly distributed and concurrent document editing with Operational Transformations (OT) versus with Convergent Replicated Data Types (CRDTs). 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.

The custom CRDT on which the collaborative text editor is based is described in detail in §3.1. In contrast to the OT-based library ShareJS, my system also 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 architecture. The simulation allows me to control the precise topology, link latencies, and protocol and explore advantages and disadvantages of using a P2P approach.

One extension, adding undo functionality to the CRDT, was also completed. I developed two approaches implementing different semantics and consistency models originally, before reading related literature. However, one paper, Logoot-Undo, takes a very similar approach and is discussed briefly below.

1.3   Related Work

Part of the challenge of this project was to develop my CRDT and associated algorithms based only on a high level explanation of the desired functionality provided by Martin Kleppmann. As a result, my solution is not optimal in all aspects, and could be improved upon in the future. Several possible optimizations are discussed throughout chapter 3. It also falls into the class of *tombstone* CRDTs, marking elements as deleted rather than fully removing them, which forces the data structures to grow continuously until garbage collected. Other CRDTs are *tombstone-free* and do not suffer from this unbounded growth. Existing CRDTs of both types are discussed here.

1.3.1 Treedoc

Treedoc [12] is a replicated text buffer; an ordered set that supports insert-at and delete operations. This CRDT gets its name from the tree structure 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 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. Each insert is tagged with a unique client identifier which comes from an ordered space. Using the identifier order in combination with infix traversal creates a total ordering over the characters contained in the tree, including those grouped as mini-nodes. With the total order, all clients retrieve the same string from their Treedoc replica. Having a total order is an important property used to guarantee eventual consistency in CRDTs. Using unique, ordered client IDs to provide a total order in concurrent cases is common in CRDT design and indeed used in my own CRDT.
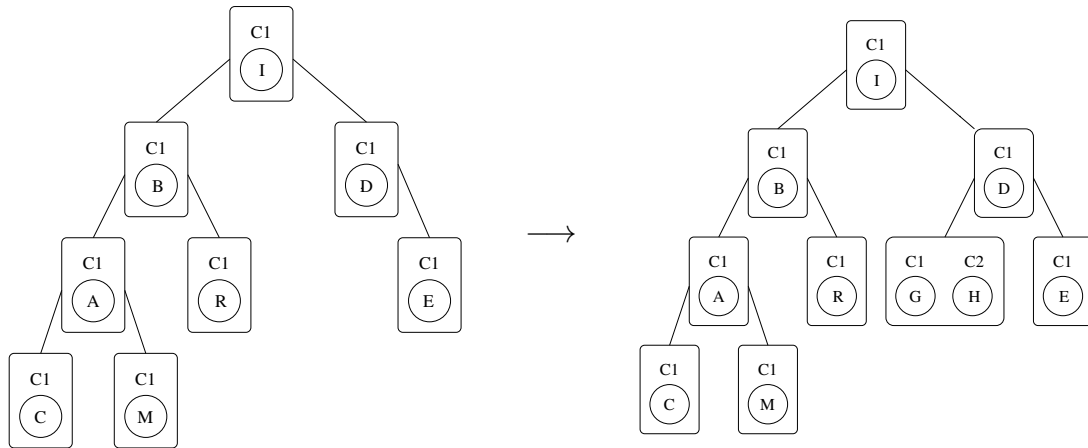
Figure 1.1: Concurrent updates to the same node. In the first state (left), client 1 has written the string 'Cambride' into the text buffer. The systems settles and client 2 also sees the string 'Cambride'. Both clients 1 and 2 realize there is a missing character – client 1 inserts 'g' and client 2 mistakenly inserts 'h'. As both create the same node in the tree, the nodes are merged as mini-nodes into a larger node. Both clients now see the string 'Cambridghe'.

Deletes in Treedoc are handled by marking a node as deleted (but the node remains in the structure). Thus Treedoc falls into the class of *tombstone*CRDTs. As deletes and inserts are not guaranteed to result in a balanced tree, the path to a node in the tree (i.e. its identifier) can become very long. The authors propose an expensive commitment protocol to rebalance it periodically. Not only is this inefficient, but also rather contrary to the spirit of consensus-free operation that makes a CRDT so useful.

1.3.2   Logoot

Logoot [20] belongs to the class of text CRDTs which do not require tombstones for dele-
tion. It achieves this by creating totally ordering 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 the identifier
and the data it tags without adverse effects on the rest of the CRDT.

This method 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 [11] [10] 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 very simple and optimal identifier scheme.

Further research with Logoot enabled 'undo' and 'redo' functionality, which is described
below.

1.3.3   Logoot-Undo

CRDTs generally struggle to provide an undo mechanism since the concept of reversing
an update to the data structure is fundamentally contrary to the key property of CRDTs:
commutativity of operations. For example, removing an insert does not commute with
the original insertion. If it were, the insertion of a character followed by its removal
would produce the same result as removal followed by its insertion. In the first case, the
character is removed. In the second case, the removal has no effect since the character
does not exist yet, so the following insertion takes effect. Thus the order the operations
take effect matters; they do not commute.

Logoot Undo [19] proposes to resolve this by essentially tagging each identifier with a
visible counter. An undo of an insertion would decrement it, while redo would increment
it. If the *visible* counter is positive, the characters are visible. 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, where clients can undo anyone's operations.

Chapter 2

Preparation

2.1 Consistency Models

2.1.1 What is "Conflict Free"

One important definition is the exact meaning of "conflict free". There appear to be multiple interpretations. 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. 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. Conflict free then means that no data is lost, and after all operations are exchanged the resulting states agree.

To demonstration, two common conflicting operations in text editing are inserting characters into the same index (on different clients) and simultaneously deleting the same character. The second is straightforward to make conflict-free, and both the user and data oriented definitions of conflict agree – deleting a character on a single user system or multiple times concurrently should still result in the character disappearing. In the case of inserting text into the same index, both users expect their own text to appear in the index they inserted at. If this were carried out, one write 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 at the desired index, and shift the other user's characters to appear after. Both operational transformations and CRDTs achieve this in fundamentally different ways. This entire process is demonstrated in 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 [19].
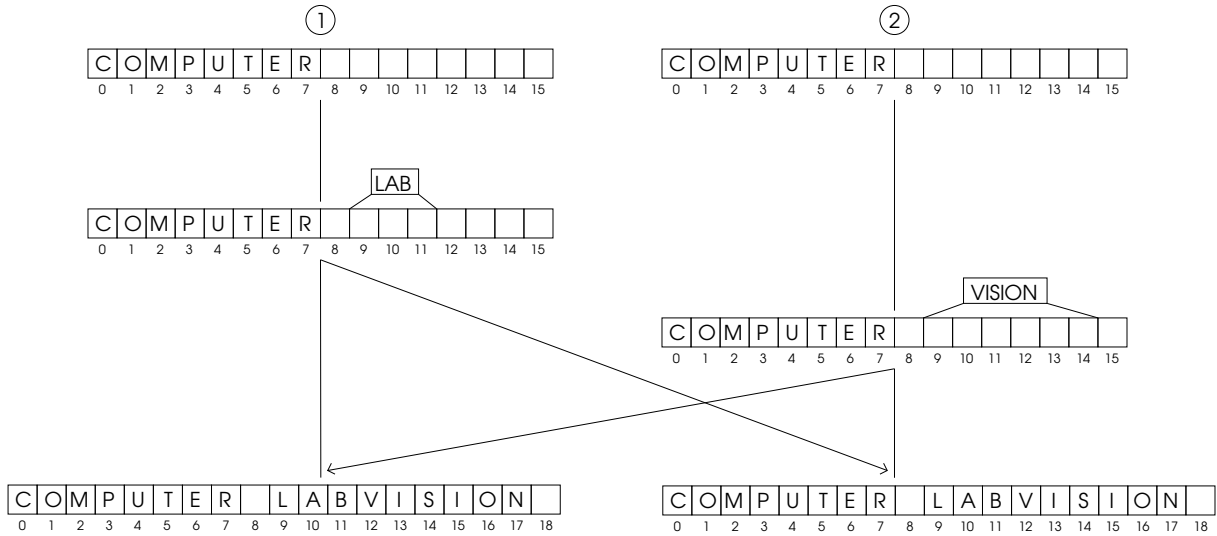
Figure 2.1: Two clients are initially in a quiescent state i.e. the system has settled with the shared string 'computer '. They then concurrently insert different words at the same index. At first each sees only their own edit. Then operations are exchanged and the system reaches quiescence again. Both clients see the string 'computer labvision'. Client 1 thus 'won' and kept the original index, while Client 2 had its insertion offset.

- **Consistency:** All operations ordered by a precedence relation, such as Lamports happened-before relation [9], are executed in the same order on every replica.

- **Convergence:** The system converges if all replicas are identical when the system is idle.

- **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; the order relation between the document lines and a newly inserted line must be preserved on every peer.

  - *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 accepted, but may produce some unexpected results as seen when implementing Undo functionality in §3.4.3.

2.2   Achieving Eventual Consistency

As mentioned briefly in the prior section, operational transformations and CRDTs aim to achieve eventual convergence on all clients. The common conflicting operations that must

be given special consideration are concurrently inserting characters at the same index, deleting the same character, and deleting a character while removing characters before it.

2.2.1 Operational Transformations

The easiest way to understand operational transformations is by example.

Figure 2.2 demonstrates the *concurrent insert at the same index* case.
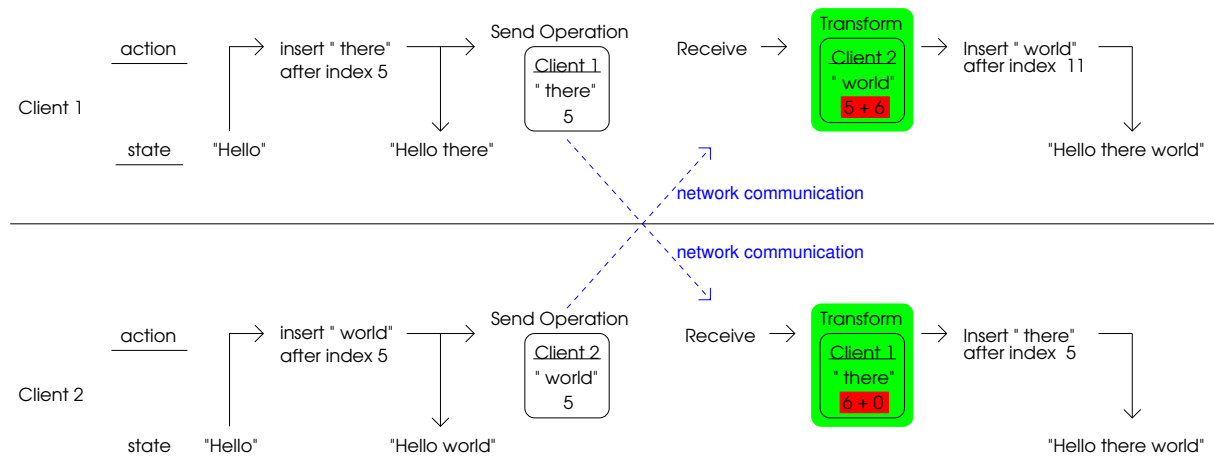
Figure 2.2: This figure shows how operational transformations might handle concurrent insertion at the same index. Here, both clients insert different strings after index 5. The operations are exchanged, and the key transform function (in green) detects the conflict, and chooses to offset " world" on 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"

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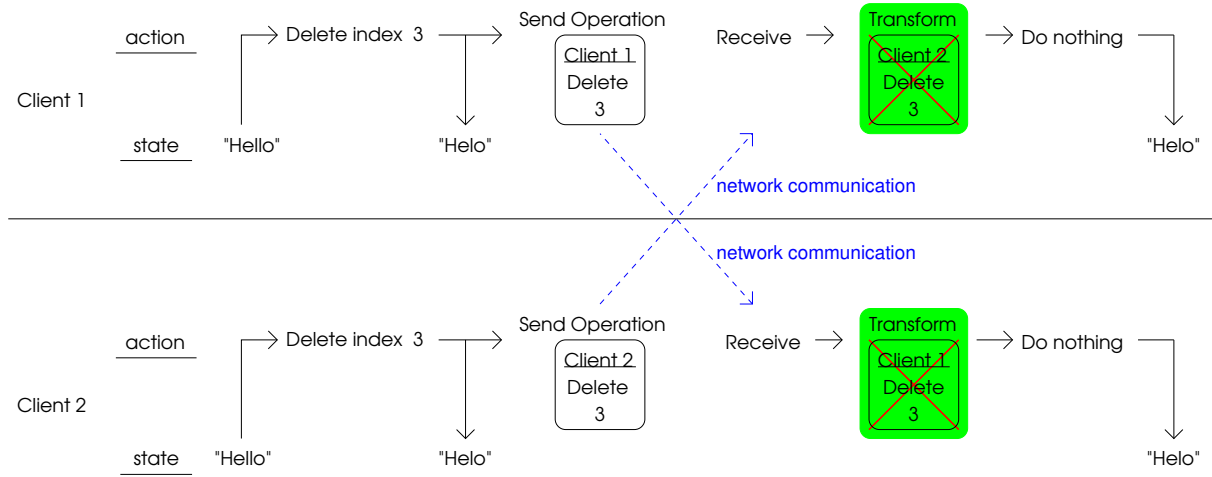
Figure 2.3: Both clients concurrently delete index 3 of "Hello", resulting in "Helo". The operations are exchanged. The transform function (in green) detects the conflict, and on both clients discards the remote operation. Integrating it would cause modifications the user did not execute (i.e. delete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

Figure 2.4 demonstrates the *concurrent insertion and deletion* case.
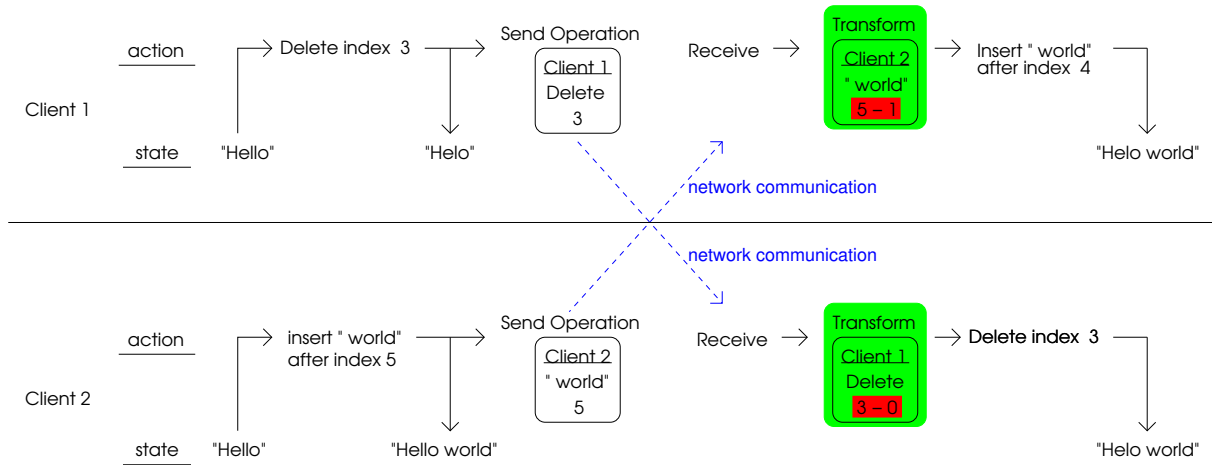
Figure 2.4: Here Client 1 deletes index 3 of the initial string "Hello", resulting in "Helo". Meanwhile, Client 2 inserts " world" after index 5 resulting in "Hello world". The operations are exchanged. The transform function (in green) detects the conflict, and on Client 1 knows to subtract 1 from the 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   Convergent Replicated Data Types

This section will provide an intuition for CRDTs for text editing in general, while the specific CRDT used for this project is outlined in §3.1.

CRDTs, which were first formalized in a 2007 paper [13], trade the complex algorithms used in OT for a more complex data structure. Rather than relying on a serial order provided by a server, or logic to transform operations against each other, data are tagged with totally ordered identifiers which allow us to extract the data in the native form – for example, a string will be represented as a set of tagged characters, so they may be read out according to the tag ordering. Figure 2.5 is a simple demonstration of how this works.
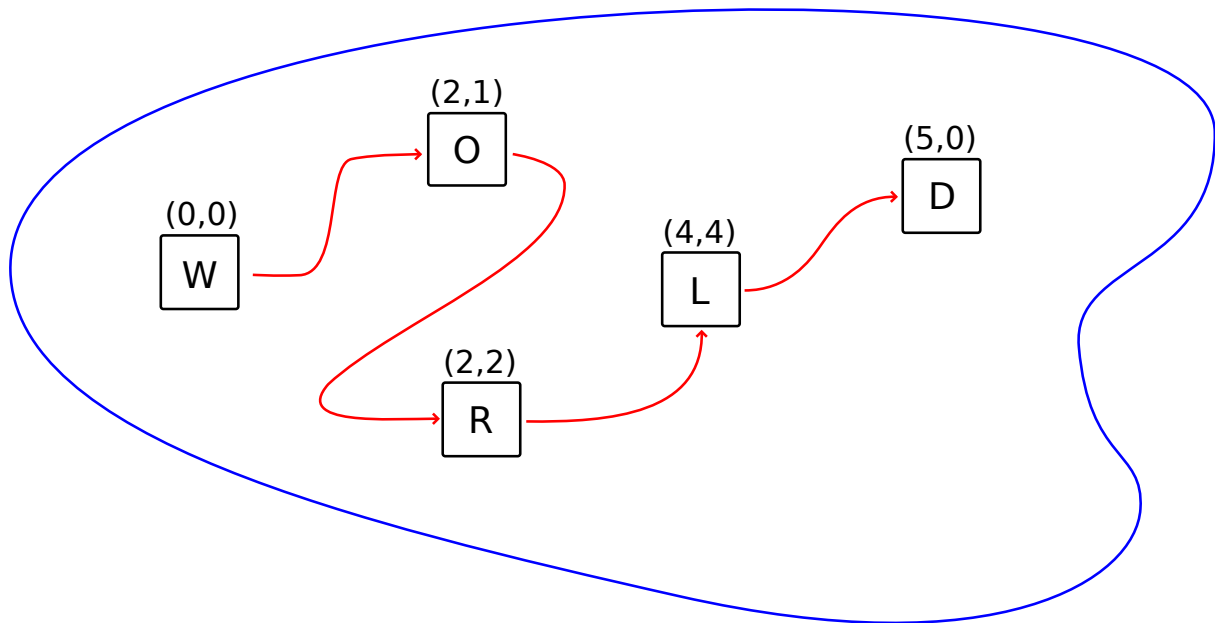
Figure 2.5: A CRDT containing the word "World". The CRDT is a set of nodes that are tagged with ordered identifiers. The order used here is <, ordered by the first element of the pair, then by the second. The red arrows trace out the ordering, which presents the word "World".

There are technically 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 [14]. Operation-based CRDTs relay modifications to other clients, which execute them on the local replica. These only require that all operations commute, and that the communication layer guarantees only-once, in order delivery [16]. This project uses an operation-based CRDT, so the key property to fulfill is commutativity of operations while providing network layer delivery guarantees.

If these requirements are met, all clients will to converge to an identical, ordered result. This follows from the fact that elements in the CRDT have a total order defined over them: as long as all modifications arrive intact, all clients can retrieve the correct data.

2.2.3 ShareJS

ShareJS [6] is an open source Javascript library implementing Operational Transformations which can be deployed on web browsers or NodeJS[1] clients. It is useful to know more precisely how ShareJS operates and what kind of behavior might be expected for later analysis. As there are a large variety of algorithms that can enable OT [8], 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 supported actions *insert* and *delete*.

Each operation applies to a specific document version. The version number is used to transform operations against each other and detect concurrent changes on the server. The supported operations are insert and delete, and the resulting modifications are sent as JSON to the server.

An Insert operation for adding text at index 100 in document version 1:

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

A deletion of the world "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}]}
```

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 has an effect on memory consumption. This is confirmed in [TODO reference needed to experiment].

ShareJS clients can also only have one packet to be in flight to a 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 send decreases and packet size increases. Additionally, since the server can reject operations that were generated and applied at a client, the clients must be able to undo rejected operations, as well as manage any subsequent, dependent operations that have occurred (this is one of the key parts of *transformation* of operations). To enable this, the clients must each have a list of past operations, which also affects memory use [reference to experiment]. The need to maintain a list of past operations plus the document of n characters generated by at most n operations suggests a ShareJS client's space complexity might be $\Theta(n)$.

---

[1]https://nodejs.org/en/

2.3 Starting Point

As stated in the proposal, I had prior experience with ShareJS, which was leveraged when creating the comparative system. Additionally, I was already proficient in Javascript and had working knowledge of Typescript, my main implementation language. However, almost all other aspects were new, notably: learning about CRDTs, writing test cases, the process of creating experiments and using these to profile performance, and how to implement a simulation.

As the project progressed, several courses contributed or reaffirmed ideas I could use. Notably, the Computer Systems Modeling [3] course had a short 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 [18] provided valuable background towards Lamport and Vector clocks, causality, and total orderings in distributed systems; the IB Computer Networking course [2] gave me a foundation and overview helpful 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 the CRDT based systems

Points one and three have multiple unspecified subgoals. For clarity, Table 2.1 lists these and their respective importance and difficulty. The goals closely mirror the 'Detailed Project Structure' of the proposal.

Table 2.1: Project 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

2.5   Software Engineering

2.5.1   Libraries

ShareJS [6] is the main external resource I required. It is released under the MIT license. I used the simpler ShareJS v0.6.3 rather than the more current ShareJS 0.7, also known as ShareDB. This package was installed via the NPM[2] package manager. The other large library I used was D3.js[3], a commonly used data visualization tool that helped me build a dynamic network graph for debugging purposes. I did a survey of other drawing libraries that might be simpler and lighter on resources, however in terms of documentation, ease of use, and familiarity I did not find anything more suitable.

2.5.2   Languages

The three main implementation languages, by lines of code, are Typescript[4], Python 2.7[5], and Coffeescript/Javascript (mainly in ShareJS). Reasons for choosing Typescript as the primary language are familiarity, how easily it integrates with web technologies and JSON objects, typing – which helps with project scale and early error detection –, and the fact that ShareJS ships as Javascript, which Typescript transpiles to. In order to maximize code reuse and comparability of results, it makes sense to run both systems on the same execution platform, discussed next.

2.5.3   Tooling

The testing platform needs to be a web browser or NodeJS for compatibility with ShareJS. I chose to use a browser due to familiarity and possibility of providing this work as an open source library in the future: it has a wider reach with browsers. The most developer friendly choices are Mozilla Firefox[6] and Google Chrome[7], as both come with sophisticated debuggers and script inspection capabilities. However, both have issues for this project. Firstly, measuring API for measuring memory consumption in Firefox is complex and badly documented[8]. On the other hand, Chrome offers a simple interface to measure memory when certain flags are enabled. Conversely, I discovered Chrome does not allow more than 6 active TCP sessions to a single domain from one session, which I needed to do when running an experiment with more than 6 clients in a single browser tab. Firefox has a simple *about:config* setting where this limit can be increased. Luckily, ShareJS

---

[2]https://www.npmjs.com/

[3]https://d3js.org/

[4]http://www.typescriptlang.org/

[5]https://www.python.org/

[6]https://www.mozilla.org/en-US/firefox/new/

[7]https://www.google.com/chrome/

[8]https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Reference/Interface/nsIMemoryReporterManager

contains a built in workaround for the TCP limit most browsers have. Thus with memory measurement support and a solution to the TCP limit, my platform of choice is Google Chrome version 56.

Before starting this project, I was already familiar with a specific Typescript development stack and environment. The wide range of choice available for web development work flows pushed me to use what I was already somewhat familiar with. This includes package manager NPM, Typescript, transpiler Babel, and script bundler Webpack, while coding in Visual Studio Code, an open source IDE largely developed alongside 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.

2.5.4 Backup Strategy and Development Machine

Backups and data safety were mentioned in the project proposal. Github[9] provided the primary backup, with commits at important checkpoints and at least once per work day. The local repository is also stored in my Dropbox folder for continuous cloud backups. To prevent data loss in event of operating system failure, the primary development OS Ubuntu 14.04 LTS x64 resides on its own hard drive, separate from user data. A backup development environment, Windows 10, exists on yet another hard drive. The MCS computers are the alternative in case of loss of laptop.

2.6 Early Design Decisions

From the outset, I knew I could make simplifications in some aspects of the project, and would likely need to be more flexible and verbose in others. These design decisions were made at various points throughout the development process, though happily most were made early on and required little subsequent change.

2.6.1 Network Simulation

One broad category of decisions has to do with the network simulation I implemented. Because I had no experience with simulation design and networking is not the intended focal point of this project, to begin with 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. We will see how to relax some of these assumptions in §3.1.5. Broadcast is not typically found in Internet applications. For instance, IPv6 phases out broadcast functionality and opts for multicast instead [1]. Using global broadcast, or flooding, has severe implications in terms of network efficiency. Without further measures, basic flooding sends $\Theta(n^2)$ packets, where n is the number of clients in a fully connected network.

---

[9]`github.com`

This property can be seen in the Evaluation section [experiment ref]. However, though it has downsides, broadcast is simple to simulate on given a network topology, requires no addressing, and no sophisticated protocols.

While the broadcast is a useful simplification, the topology of a P2P network affects a system's functionality nearly as strongly. As this project is somewhat a comparison between P2P and client-server architecture, being able to run experiments over different topologies is important. My initial focus was on a fully connected P2P topology to contrast with the client/server star topology. However, forcing the P2P simulation to run on a star itself is perhaps a more direct comparison. With two topologies to test it is already sensible to have a fully general mechanism for specifying a network, so I chose to provide support for arbitrary topologies and latencies on individual links.

2.6.2   Data Collection and Logging

The 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 stringified using a standard JSON object to string conversion. This allows fair comparisons working across platforms, and is the most obvious way to measure the size of a JSON object. Additionally, ShareJS utilizes JSON. Alternatives exist which provide more efficient serializations, such as Protocol Buffers [**protobuf**]. However, utilizing a more efficient serialization than ShareJS would make for unfair comparisons. Additionally, since most packets sent into the network are small, JSON overheads are relatively small. For large transmissions however, such as state replay of a CRDT §3.1.5, they would be worth considering.

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, I log payloads of packets from within the applications. This is the fairest to do comparisons between a simulation's network traffic, whose packets contain no headers or other overhead, and a real TCP/IP stack's traffic.

Chapter 3

Implementation

The chapter describes the implementation of both real time editing systems, firstly the one based on CRDTs and secondly the one based on ShareJS, the experiment generation and results analysis components that are shared by both systems, and lastly the extension that adds Local Undo capability to the CRDT.

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 network stack hides from the client that the network is simulated – in the background, a large part of the work is handed off to the network simulation manager. This approach aims to ease exchanging the simulation for a peer to peer protocol such as WebRTC[1] at a later date. It also allowed 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 to the network to transmit 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.

---

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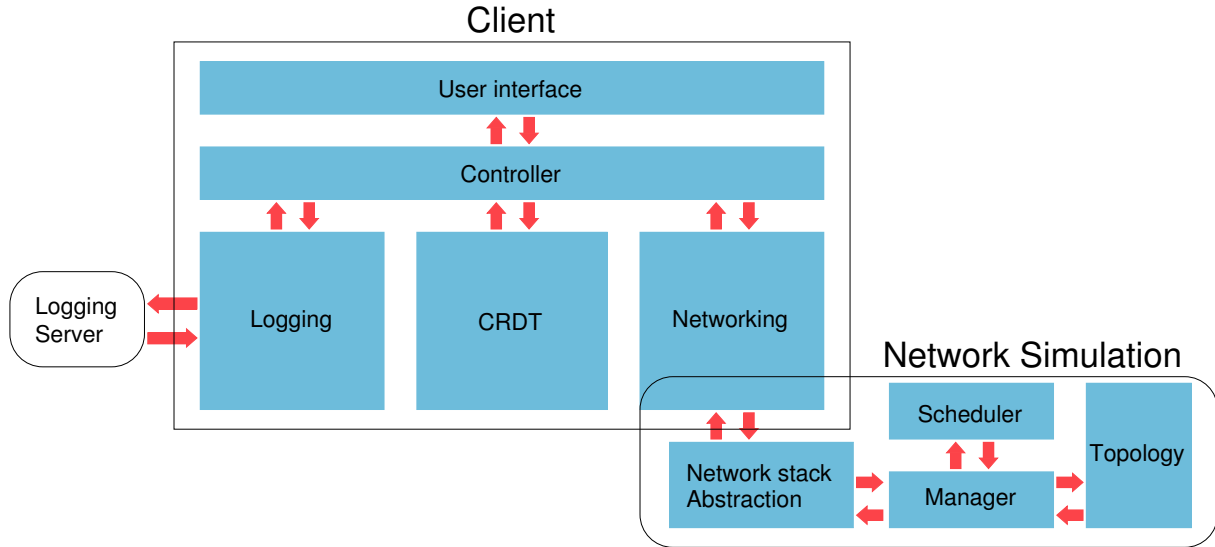
Figure 3.1: An overview of the system architecture implemented for the CRDT-based text editor. Each client has its own UI, controller, CRDT, logger, and networking component. The network module abstracts away the underlying simulation. Many clients can connect to the simulation network abstraction and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

3.1.2   CRDT

To briefly review §2.2.2, a text CRDT can be thought of as a set containing characters tagged with totally ordered identifiers. The document is then extracted in full by ordering the elements according to the total order.

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 §1.3 various structures 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 implements a singly-linked list. Each link contains exactly 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
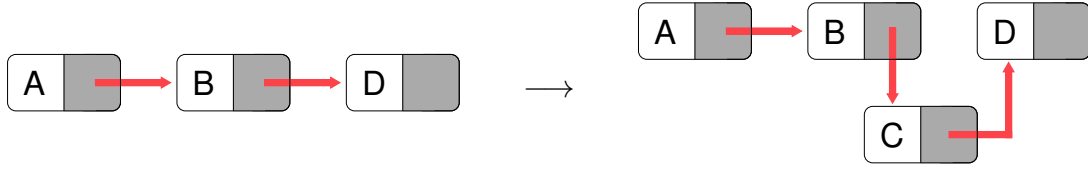
Figure 3.2: A graphical representation of updating linked lists. Here, a new link containing the character 'c' is inserted between 'b' and 'd' to produce 'abcd'.

As a brief explanation, insertion is handled similarly to insertion into a standard linked list. Deletes add to the relevant link a 'deleted' tag. Lastly, the read operation is a linear time traversal over the linked list, beginning with an invisible anchor element that marks the start of the document and cannot be deleted.

Various data structures were considered when implementing this linked list. The most 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 a linked list are $\Theta(n)$: finding a node to insert after or delete requires scanning some proportion of the list. A read is $\Theta(n)$.

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

```
 1  {
 2      "0":    {"n":"7.1","c":""},
 3      "1.0":    {"c":" ","n":"2.0"},
 4      "2.0":    {"c":"w","n":"3.0"},
 5      "3.0":    {"c":"o","n":"4.0"},
 6      "4.0":    {"c":"r","n":"5.0"},
 7      "5.0":    {"c":"l","n":"6.0"},
 8      "6.0":    {"c":"d","n":null},
 9      "7.1":    {"c":"H","n":"8.1"},
10      "8.1":    {"c":"e","n":"9.1"},
11      "9.1":    {"c":"l","n":"10.1"},
12      "10.1":  {"c":"l","n":"11.1"},
13      "11.1":  {"c":"o","n":"1.0"}
14  }
```

Figure 3.3: A sample CRDT annotated with arrows. These denote the links implemented within the has hash map. The anchor element tagged '0' is invariant and is invisible to the user. The final link, tagged "6.0" has no further pointer. Pointers are labeled n. The overall string is "Hello world".

Javascript provides two native objects capable of mapping: Map and standard Javascript Objects. I implemented my CRDT using both structures and eventually settled on native Objects for ease of serialization to JSON and the inconvenience of using Maps – Maps hash pointers to keys rather than processing the keys themselves, making them difficult to work with.

From now on, "CRDT" and "linked-list" will be used interchangeably.

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 *cid*s.

Define an identifier generated by client i to be a pair (t_i, cid_i) where t_i is a the value of a local counter incremented on every insert. cid_i is the globally unique identifier of the client. 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 [9]. If an incoming operation has a greater t_j than the local clock, the value of t_i is set to t_j. This guarantees that any operation that is causally generated after another (defined in more detail later) has a higher clock value, which in turn guarantees that only concurrent operations can have the same clock value. Note that this does not mean all concurrent operations have the same clock value. The only meaningful deduction we can make is that a remote, incoming operation that has a lower t than the local clock, must have been concurrent. This is a key idea utilized when inserting characters concurrently.

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

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

Operations

The text CRDT supports two core modifying operations: insert and delete.

Insert   Inserting a character into the text document generates an insert operation. An insert operation is stored and transmitted as a bundle.

An insert bundle B contains

- A unique identifier *id* for the character
- The character *char* itself

- The node after which to insert the character denoted $after$. This corresponds to the position in the linked list at which the character needs to be spliced in. $after$ is another unique identifier

```
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 follows 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 what order the concurrent inserts are incorporated.

To understand the intuition behind why this works, recall the reason for utilizing a Lamport clock: we can deduce that a remote, incoming identifier id_a and some locally generated id_b, are concurrent if $id_a <_{id} id_b$. Thus, when incorporating id_a, we skip over local, concurrent identifiers until finding one where the condition $<_{id}$ fails and insert there. Reciprocally, once local operations arrive at the sender, the same algorithm will be applied. The arrived operations will have greater identifiers, $id_b >_{id} id_a$ (same identifiers of concern as before, just on the other client), and so step 2 will iterate over nothing and insert them before id_a. Thus on both clients, id_a will be reside after id_b in the linked list.

Delete   Deleting a character from the text document generates a delete operation, which is transmitted as a bundle B 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 into the CRDT is straightforward

1. Locate the node N corresponding to $B.deleteId$

2. Set a boolean flag in $N.d$ to true

3.1.3   Tombstones

The delete operation described previously never removes nodes, but leaves them behind as tombstones. Some CRDTs, such as Logoot described in §1.3.2, are structured such that the document is 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 our 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 very high latency link.

The process of establishing that each client has received and executed an operation can be achieved using an expensive commitment protocol, which is what is suggested in [12]. In effect, the system periodically executes a distributed garbage collection. While possible, I have not implemented this protocol. Removing tombstones may be necessary after the data structure becomes too large, but until such a point tombstones are useful in implementing undo functionality for the text editor. This is discussed in section §3.4.

3.1.4   Optimizations

The insert operation produces a bundle which contains exactly one character, one identifier, and one *after* tag. We can drastically cut the number of operations generated, and thus packets sent in the network, by allowing an insert bundle to contain a contiguous sequence of characters.

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 receiver 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 the (t_{after}, c) *after* pointer and each new node points to $(t + (i + 1), c)$. The final node points to the original target of $(t_{after}, c$. The resulting size of the CRDT is same as an optimized bundle: this is a network only optimization.

This insert optimization has potentially massive gains in terms of 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 protocol can send arbitrarily long packets, the application-layer network capacity requirement is reduced to between $\Omega(n)$ and $O(nlog(n))$ from $\Theta(nlog(n))$. Written another way, if an

average of m characters is sent per packet, the capacity needed is $\Theta((m+log(n))*n/m) = \Theta(n+nlog(n)/m)$. The number of packets is reduced to $\Theta(n/m)$ which is desirable since standard protocols such as TCP have high per-packet dissemination overheads.

Another optimization I added was renaming tags and names to be as short as possible (often single characters), so that the resulting serialized JSON string sent over the network is as short as possible. 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 fair.

3.1.5 Network simulation

The section describes the network simulation that delivers operation bundles from one client to another. First, I will detail the initial assumptions I 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 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 continuity I will continue with *simulation*.

Assumptions

The CRDT I implemented requires that messages be delivered causally [18]. We define the happens-before relation $a \to b$ to be true whenever a happens before b on the same process, for example

$$Receive\ Insert\ ``Hello'' \to Insert\ ``World''after``Hello''$$

We then require that all events ordered by $\to$ be delivered in a valid ordering according to $\to$. We call this "Causal Order". This defines a partial order, since there may be some A and B such that neither $A \to B$ nor $B \to A$ 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.

Guarantee of Causal Delivery. 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 on any shortest path p from $sender_B$ to another node on p. Denote the ith node on p as p_i. Use i as the induction variable. 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. Thus, the network guarantees causally ordered delivery to every node in the network.

$\square$

That the network is able to guarantee causal delivery is a strong assumption and cannot be made in general. We will see in section [section ref] how to relax network assumptions 1 and 2.

Abstraction

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 a copy. The Network Interface essentially emulates the top of a classic network stack, whereas the manager abstracts away the bottom layers.

Network Interface   The essential parts of the Network Interface type signature are shown below.

```
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 (cleaned)

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 CRDT or 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 and requesting a copy of the CRDT be sent from an active client.

Joining the Network   To make the system more flexible, I added the capability to join (but not leave) the P2P network during execution. Only the first client gets to create a new CRDT; joining clients request a copy of the CRDT via `RequestCRDTMessage` (see prior section). The response `ReturnCRDTMessage` can be quite large – the later a client joins, the larger the CRDT, and the larger the packet. Though not implemented, as many of the characters transmitted may be redundant JSON delimiters, protocol buffers [**protobuf**] could optimize the size of these messages.

An alternative method to requesting an up-to-date CRDT is to begin with a empty CRDT and replay all subsequent operations on it. This would however be even less efficient as it would require all remote clients to store all of their previous operations forever (or have a server store them), and local 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 with my implementation. If a client somehow has an out of date CRDT, it must request an entire new copy via `RequestCRDTMessage`.

At this point it is important to acknowledge that introducing dynamic network joining violates one of the guarantees of §3.1.5. 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 (**??**) will restore this guarantee.

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 (discussed further in section [section ref]). 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 sending copies of CRDTs. Overall, this module replaces the network and data layer of most network stacks and abstracts away how packets get exchanged between neighboring clients.

Scheduler

Although this subsection falls under the Network Simulation section of this document, the scheduler is an altogether more general driver of the simulation. However, its main task during an experiment is to deliver packets between nodes, which is why it is listed here.

A simulation scheduler is responsible for mutating system state, based on events given to it 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

My 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 packets onto a single link at the same logical time, and §3.1.5 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 [3, slide 120] course, is that the next executed event be the one with the least remaining time. Using a heap, we get $\Theta(log(n))$ retrieval per element. When the simulation is running, the scheduler removes the top event E off the heap, decreases all remaining events' primary keys by $E.pKey$, and executes $E.payload$. This may in turn generate more events which are added back into the heap.

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

Time The concept of time in a simulation is generally taken to be "logical time". The system begins in $t = 0$, and each subsequent event moves the t variable forward. This works perfectly well for the CRDT-based system, 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 will have very small deltas, for which starting and stopping a timer would be impossible.

In this project, I implemented both an event-driven scheduler and a timer-driven scheduler. The timer-driven version is useful when 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 Δt until the next event. As the event-driven version is more flexible, and simpler – the driver is 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.

Causal Delivery

Until this point, the network has guaranteed causal delivery of packets based on very strong assumptions and knowledge of the system implementation. We can relax these to allow out of order delivery and a dynamic network topology. This can be done by ensuring causal delivery using vector clocks [5].

There is now an additional network layer, as depicted in 3.4. We now 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 $A \rightarrow B \Rightarrow deliver(A), deliver(B)$.
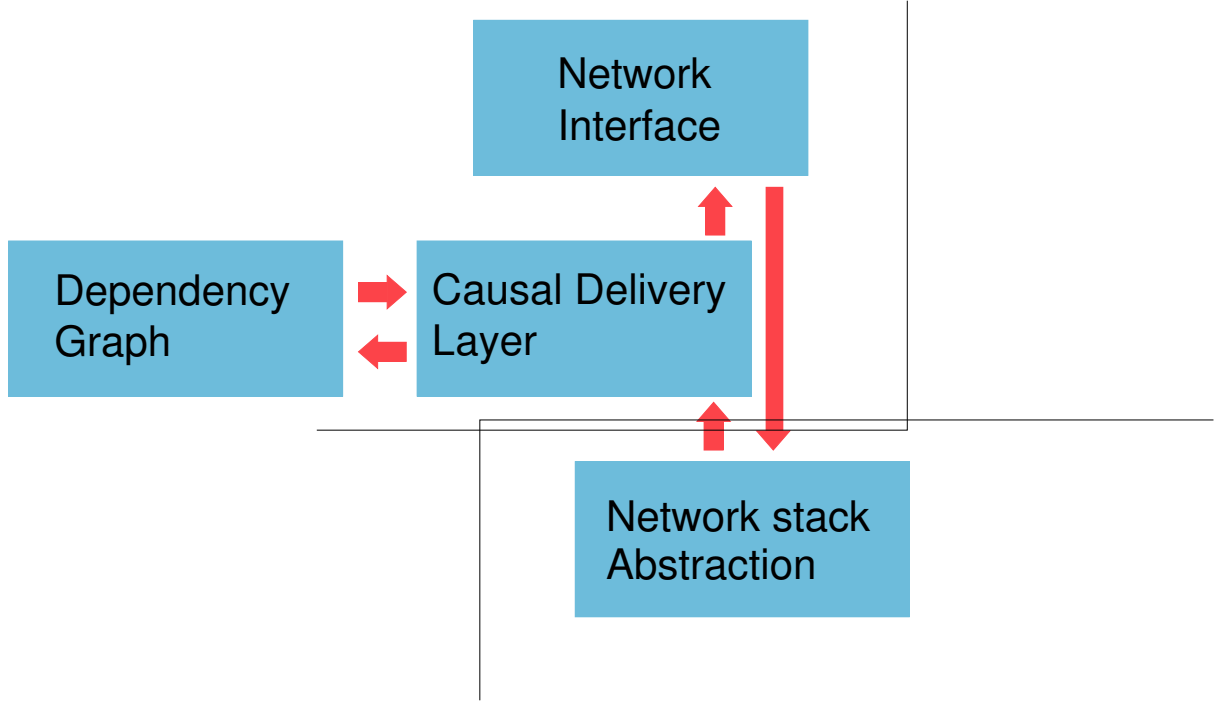
Figure 3.4

Modified Vector Clocks A traditional vector clock is outlined is 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. 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 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 define

$$v_1 \ <_v \ v_2 \iff \forall c \in v_2, v_2. \ v1.c \leq v_2.c \ \wedge \ \exists c' \in v_1, v_2. \ v_1.c' < v_2.c'$$

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 $v <_v s$, then the message has been seen before and can be discarded. If $s <_v v$, 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.

Using a delta we can more efficiently deliver buffered packets than doing a linear search through the queued packets on each new arrival. This idea is depicted in Figure 3.5.

Packets Queued From Destination ID

| 1 | | 0 | | 2 |

1 → ... 1 ... 3 → ... 2

| packet | | packet | | packet |

1 ↓
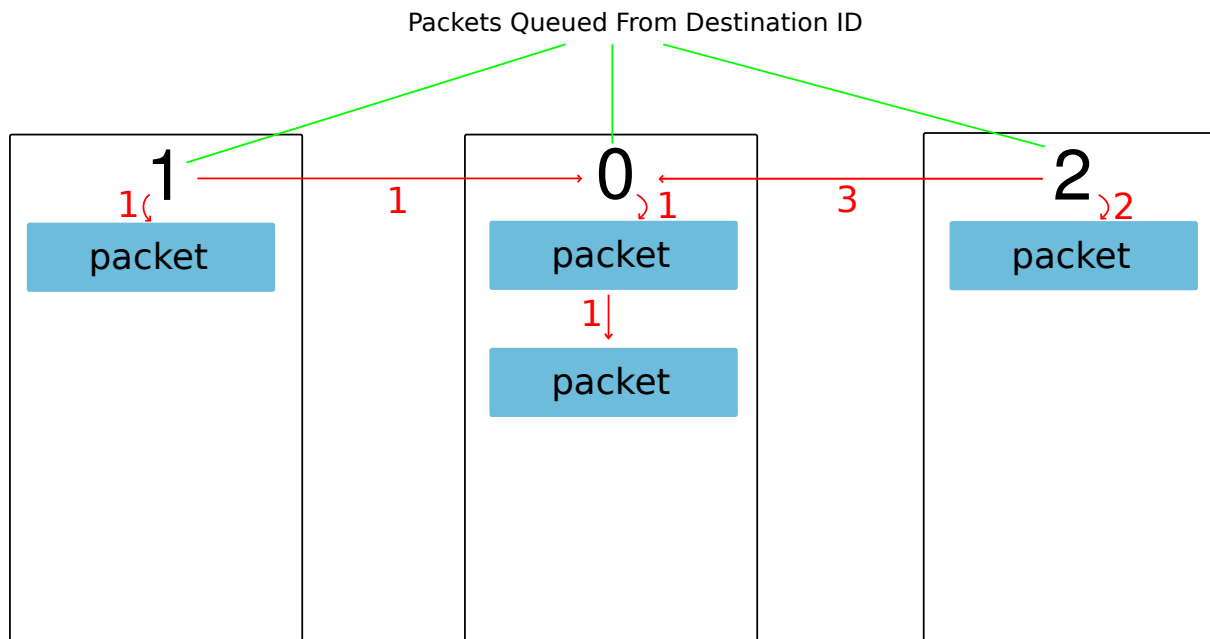
| | | packet | | |

Figure 3.5: A representation of the graph structure used to efficiently deliver packets. When a new packet arrives from some destination D, all outgoing links from D are decremented. The target of each link checks the head of its queue for potential delivery. The efficiency gain relies on the fact that messages from the same client must be delivered sequentially.

3.2 ShareJS Comparative Environment

The main difficulty when building the ShareJS-based comparative system was adapting it to allow executing the same format experiments as the CRDT-based system. This meant incorporating the real-time scheduler discussed in §3.1.5, and inserting log statements to record data to the logging server.

Scheduling An experiment consists, at its most basic, of a set of simulated events that each client performs at given 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 a logical time driven 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 difficulty was found, except the lack of type annotations in ShareJS.

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.

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 and variant is queued to the requester, which may be the CRDT-based system, or the ShareJS-based one. In either case, the same basic data is served. The system then runs the experiment and submits the resulting log back to a logging server that writes the log to the correct file system directory. 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 server, if necessary, are relaunched to improve comparativeness of the data gathered.

3.3.2   Experiment Design

A basic experiment must define the number of clients participating, a set of events for each client to execute (these are seeded in the scheduler), and the network topology and link latencies. Listing 3.6 shows all of the features visible for a simple experiment.

```
1  {
2    // Experiment Name
3      "experiment_name": "experiment_1",
4    // How many clients and when they join
5      "clients": [0,0,0],
```
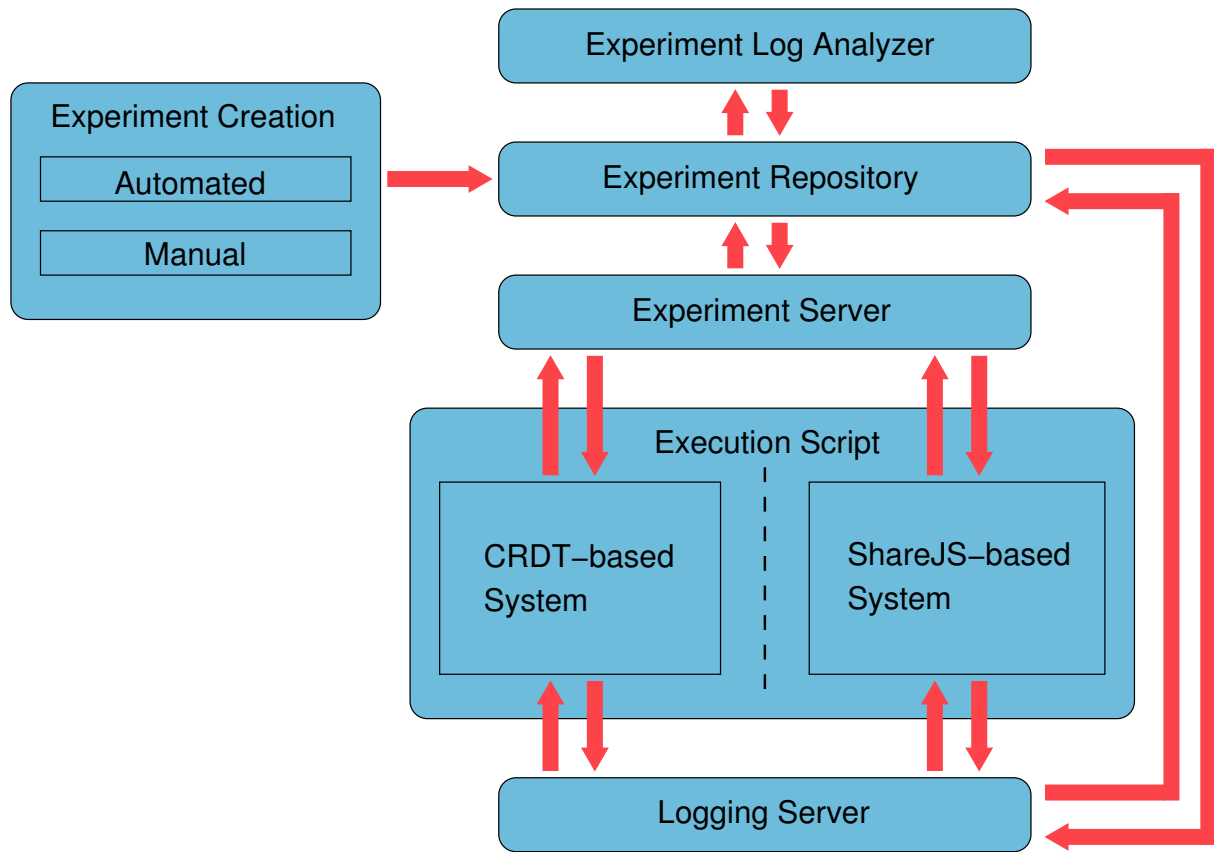
Figure 3.6

```
 6    // CRDT-only: Per client values used to compute link latencies
 7      "network": {
 8          "0": {"latency": 196.7632081023716},
 9          "1": {"latency": 220.01040150077455},
10          "2": {"latency": 122.31541467483453}
11      },
12    // CRDT-only: type of scheduler to use
13      "execution": "event-driven",
14    // Scheduled events
15      "events": {
16          "0": {                  // Events for client 0
17            "insert": {           // Insert events for client 0
18                  "0": {          // Insert "coliseums" at time 0, index 0
19                    "chars": "coliseums",
20                    "after": 0
21                  }
22            },
23            "delete": {}          // No delete events
24          },
25          "1": {                  // Events for client 1
26            "insert": {           // Insert events for client 1
27                  "15": {         // Insert "tackling" at time 15, index 0
28                    "chars": "tackling",
29                    "after": 0
30                  }
```

```
31              },
32                  "delete": {}        // No delete events
33          },
34          "2": {              // Events for client 1
35              "insert": {         // Insert events for client 1
36                  "30": {         // Insert "freshening" at time 30, index 0
37                      "chars": "freshening",
38                      "after": 0
39                  }
40              },
41                  "delete": {}        // No delete events
42          }
43  },
44  // CRDT-only: Topologies over which to run the same experiment
45      "topology": [
46          "fully-connected",
47          "star"
48      ]
49 }
```

Listing 3.6: A simple JSON experiment setup. Comments added for explanation and not part of JSON syntax

The choice to assign latencies per-node, and calculate a link latency as the average between two nodes' values, rather than assign per-link stems from the need to have one network description that fits any topology. It also models the real world better on a conceptual level: 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 Concerns

A major decision that was made was to separate log analysis from the system generating the log itself. One reason for this is that it is good software engineering: one component should do one task. Another is that a separate analysis layer can analyze results from both the ShareJS and CRDT systems, rather than duplicating functionality within each system.

Modules for tracing logged packet departures and arrivals end up resembling a set of 'clients', much like the original simulations' and ShareJS's clients. However, I decided that having a shared layer that is easily extensible is worth some high level duplication. An independently running Python script serves this purpose. A sample from a simple experiment is listed in Appendix C.1.

3.4   Extension: Local Undo

In interactive systems undo and redo are key features [15]. As such, it is an interesting extension to the capabilities of a CRDT 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 everyone can modify any operation, at the suggestion of my supervisor. The approaches detailed below were developed originally prior to reading related literature, but one is very similar to that of Logoot Undo §1.3.3.

3.4.1   Overview

Each client keeps a local stack of operations that it produced. The stack can be truncated to a certain length to avoid ever-growing memory consumption. When undoing an operation, a pointer is moved back down the stack and the inverse operation of the corresponding item on the stack is generated 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 operation on their local stack. How this concurrent case is handled leads to two different semantics and consistency models.

3.4.2   Insert

Undo and redo of an insert are not equivalent to deletion and insertion, due to the unique identifiers generated during insertion events. Rather, 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 (defined formally in §3.4.3), 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: the effect of client Bs delete returns 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 an extra boolean 'visible' tag (denoted v). The two new bundles that need to be created are undo-insert and redo-insert, which also 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, there are never conflicting operations and there is no need to prove commutativity. Operations from the originator are delivered in order by the network.

3.4.3   Delete

Undoing a delete, as mentioned before, is tricker than an insert as multiple clients can perform and therefore 'own' a delete at the same time. How this concurrency is handled determines system behavior. The two possibilities are implementing the CCI consistency model and a variant I termed "Immediate Undo".

CCI Undo

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.

Modified Links   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, nonexistant ⇒ not deleted
5    v?: boolean   // (optional) visible, nonexistant ⇒ visible
6  }
```

Listing 3.7: New Type Signature of a Link in the CRDT

Delete, Undo, Redo   Delete and redo operations now 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, only retrieve links that satisfy

$$\forall Links\ l.\ (\neg \exists l.v \lor l.v = true) \land (\neg \exists l.d \lor l.d = 0)$$

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

CCT Undo requires users agree 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   To this end, the following are the desired 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.

The follows 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 of a link $l.d$ 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.

As before, delete, undo delete and redo delete bundles contain only the identifier of the link l to act upon. In non-concurrent cases, that is, operations are causally dependent, the latest dependent operation is applied to l.

If a remote delete or redo delete operation is concurrent with the 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 links that satisfy

$$\forall Links\ l.\ (\neg \exists l.v \lor l.v = true) \land (\neg \exists l.d \lor \neg l.d[0])$$

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 are $\Theta(m * log(k))$, 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 can enable 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 (§3.1.5). 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, and so this approach is not implemented in this project.

Chapter 4

Evaluation

This chapter aims to analyze the project's success in terms of meeting the proposed goals.

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 and references to the work taken to complete them.

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

I can qualitatively assert that this first goal has been completed. Section [section ref] outlines the components needed and steps taken to create a distributed text editor based on CRDTs and connected by a network simulation. Though the primary use of the editor is to gather data via programmatically predefined experiments, it can also be used manually 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 testing results are 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. Beyond the basic implementation that will be analyzed, various optimization

possibilities (see sections [section ref] and [section ref]) were revealed while implementing the CRDT and the network simulation underlying it. These were not known during the proposal stages of the project but fall under the umbrella of criterion three. The bulk of this chapter will focus on the results obtained by experimenting on both the CRDT and ShareJS based systems, with and without optimizations. [this doesn't flow very well...]

4.2   Testing the CRDT

The testing of the core CRDT used a *black-box testing* approach [**Patton**]. 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 providing the same interface described in §3.1.2. Insert and delete operations were applied to the CRDT in different orders, and every order needed to produce the same resulting string from *CRDT.read()*.

The Typescript unit testing library tsUnit[1] provided the framework used in this process. The testing output is shown below in Figure 4.1.

Test Passed

Total tests: 18. Passed tests: 18. Failed tests: 0.

Run all tests ▶

Errors

Passing Tests

- ▶ MapCRDTTester
  - ▶ readEmptyCRDT(): OK
  - ▶ deleteUndefinedId(): OK
  - ▶ deleteDefinedId(): OK
  - ▶ insertSingleCharacter(): OK
  - ▶ insertConsecutiveTimestamps(): OK
  - ▶ insertSameTimestampsConcurrently(): OK
  - ▶ insertDelayedHigherTimestamp(): OK
  - ▶ insertOneWord(): OK
  - ▶ insertTwoWordsConcurrently(): OK
  - ▶ insertDeleteConcurrently(): OK
  - ▶ undoDelete(): OK
  - ▶ undoRedoDelete(): OK
  - ▶ deleteDeleteConcurrently(): OK
  - ▶ undoDeleteUndoDeleteConcurrently(): OK
  - ▶ deleteUndoDeleteConcurrently(): OK
  - ▶ deleteUndoRedoDeleteConcurrently(): OK
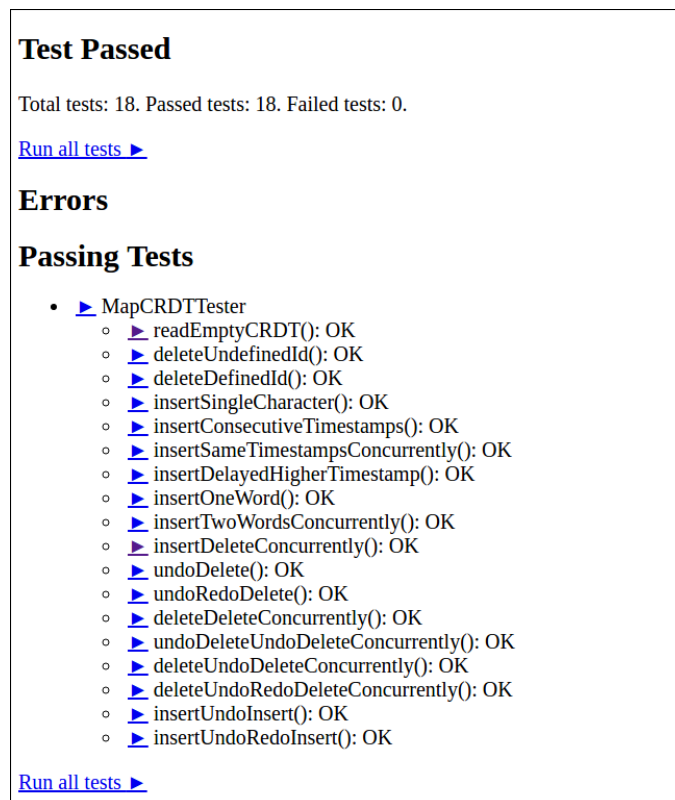 - ▶ insertUndoInsert(): OK
  - ▶ insertUndoRedoInsert(): OK

Run all tests ▶

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

---

[1]https://github.com/Steve-Fenton/tsUnit

4.3 Quantitative Analysis

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

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. Once I established that using different network topologies produced extremely similar results, I decided to combine results from experiments on both fully-connected and star topologies to better average out nondeterminism introduced by the operating system. Values are calculated as the difference from before the experiment is initialized so as to only measure the additional memory used.

Sanity Check As noted in §2.2.3 I expect 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 the document size is not growing: the only 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 [TODO] 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 prediced, whereas using undo/redo shows almost no growth.

This exposes a clear advantage for using CRDT undo/redo versus ShareJS's approach to undo/redo, which is generating opposite actions to undo a prior operation i.e. a delete to undo insertion and an insert to undo a deletion.

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
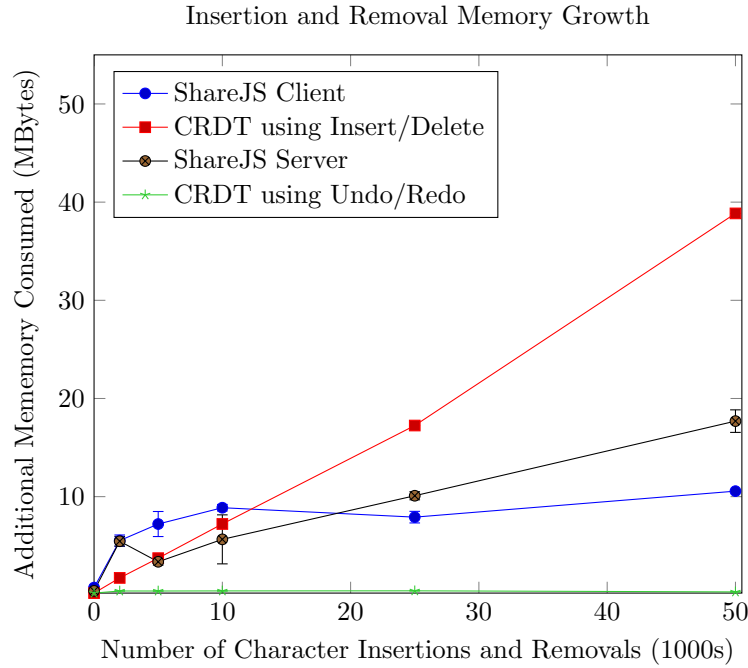
Insertion and Removal Memory Growth

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. Error in the CRDT data is generally too small to be visible.

linearly, a CRDT with n characters must store n links of constant size (ignoring deletion requirements), each identified by an identifier of decimal length $log(n)$, so one might expect a $O(nlog(n))$ space complexity. To test this, a series of experiments were designed using only a single client. Operations inserted only one character at a time, shown in 4.3a.

To compare to a more realistic workload, another series of experiments was performed inserting words of average length 7.8 at a time rather than individual characters (**??**). 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 depends on the nature of interactions with the data structures: ShareJS grows less slowly when operations are grouped into fewer packets, reflecting that it needs 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 is likely extra closures and overhead associated with a character by character operations rather than words). If operations are are kept small and frequent, CRDTs offer better performance for small and medium sized documents.

[TODO mention possible optimization of inserting into the CRDT itself in groups of splittable words. talk about in conclusion]
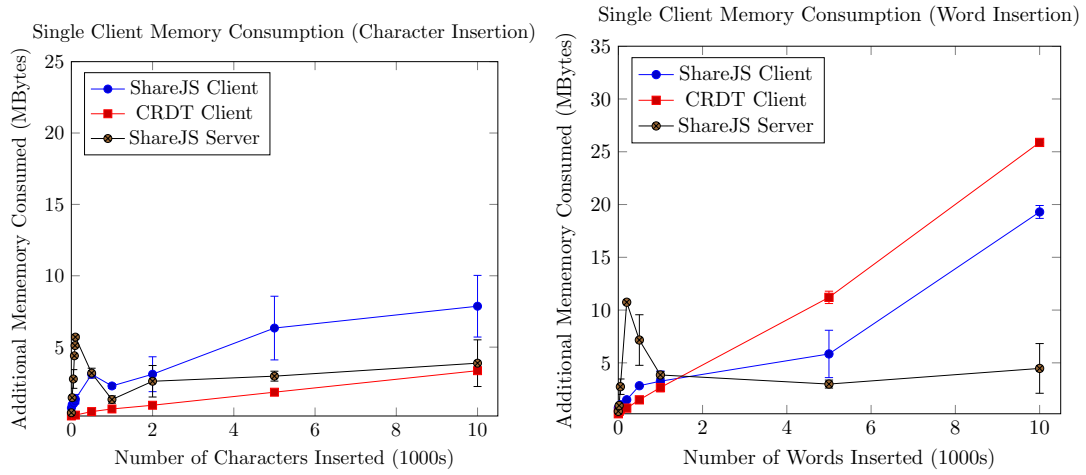
(a) A plot of component behavior as characters are inserted one at a time into the documents.

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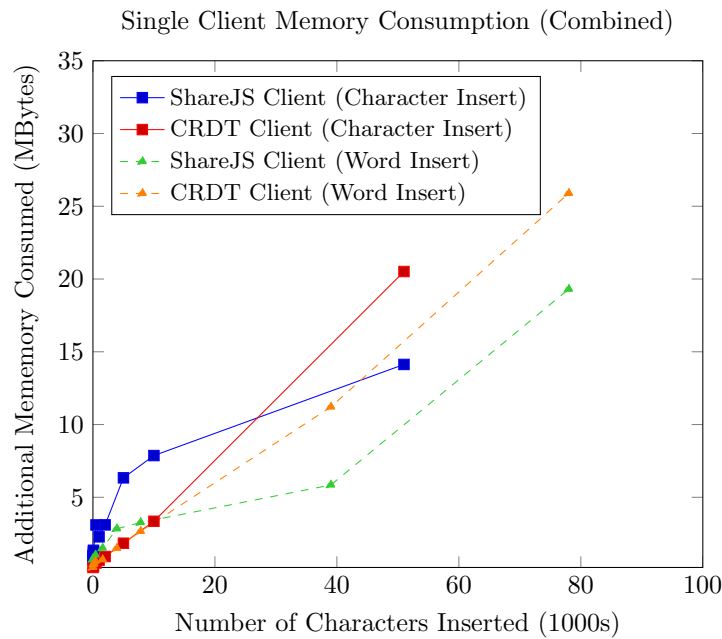
(c) A simplified plot including only the ShareJS and CRDT client consumptions on the same scale. The values from 4.3b have been rescaled by the average word length of 7.8 characters to fit the same axis as 4.3a.

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

Comparing

Drawing conclusions from these figures..... TODO

Multiple Clients   Try inserting a set number (5000) characters into the document in both systems, but with split across increasing numbers of clients. 1 Client has already been done above. The insertions were split across 1, 2, 5, 10, 50, and 100 clients. [For the CRDT system, run this on both an Undo-enabled system, CCI Undo, and the Immediate Undo system. Plot all of these on the same graph.]

Experiments 44-48 inclusive

CRDT System Variations   Though the prior data sets were collected with the arguably most useful variation presented in chapter 3, in some situations undo functionality may not be necessary. Thus a quick 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 across a mixture of insertion to deletion ratios at a fixed number of clients, operations and fixed word length.

Experiments TODO Perhaps 1000 operations per client, 5 clients at different mixes of insertion/deletion ratios, fixed length "words" of say length 5. Perhaps 100/0, 80/20, 60/40, 40/60, 20/80 is as low as we can go with 5 char words – returns to empty!. Bar graph 100/0 has already been done 1000 operations for 5 clients each in multiple clients

Experiments 49-52 inclusive, plus 45 Copy and re-evaluate for no undo and CCI Undo

4.3.2   Network

4.3.3   ShareJS Latencies

When creating fully deterministic experiments for the simulation, latencies over every link need to be specified. For realism and similarity to ShareJS, I chose to set these to be similar to the nondeterministic latencies experienced by ShareJS. I collected data on the ShareJS client to server links, and about server-client links separately.

A strange feature of these approximate distributions is the 45ms return time. I traced this into the implementation of the BrowserChannel[2] connection module used by the ShareJS client – messages actually arrive within a few milliseconds, but are only passed to the client after an additional delay of about 30-40ms.

For simulations I generate normally distributed latencies with mean 30ms and standard deviation 10ms, similar to the distributions at 200 requests per second.

Begin by empirically confirming the predictions made before...

Axes of comparison: memory, CPU, network, number of users, concurrency Network: Latency, packet size, packet quantity, cost of state replay CPU: Cost of integrating re-
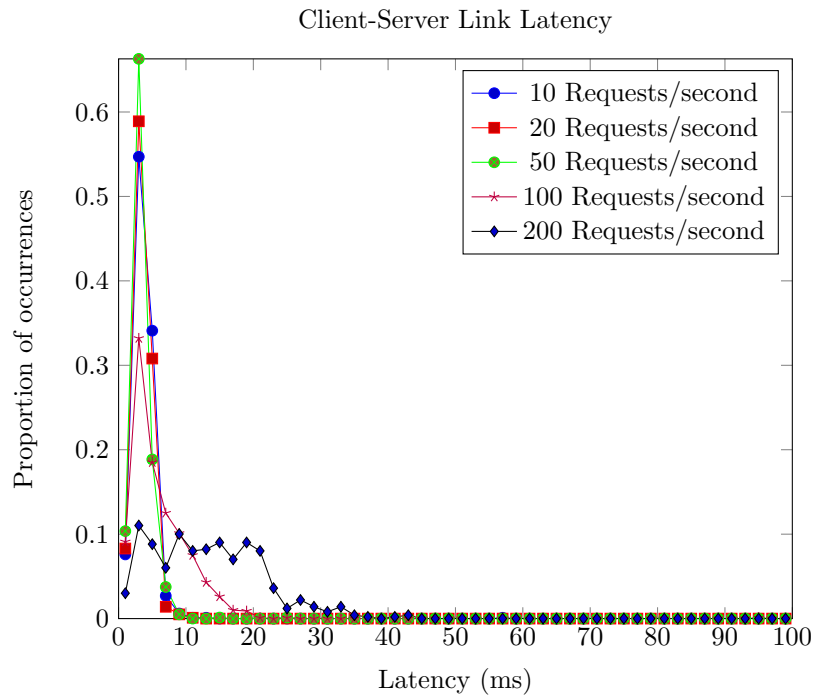
---

[2]`https://www.npmjs.com/package/browserchannel-middleware`

Figure 4.4: Latency to the server remains is typically around 3ms until a higher load of 200 requests per second.

mote changes and cost of generating local changes and sending them... how to deal with server

Lastly, a real world simulation based on user interactions with wikipedia

Experiments

CRDT: sending N words of average length k =¿ packet size growth ... cf above. Perhaps same plot?
CRDT: Flooding as a function as number of clients at one or two topologies (fully connected and star)

If have time: Implement lpbcast and anti-entropy and show latency and number of packets vs infection rate

ShareJS, CRDT: Network cost of State Replay, especially growth over size of document
Make note about protocol buffers here and the potential of saving multiples of space size. Focus on growth
Perhaps manually convert a few into proto buffers as a demonstration

CRDT: CPU cost of inserting 10000 words of length K into CRDT (easy enough to measure). Perhaps up to but not including sending into network
ShareJS: CPU cost of inserting same 10000 words into document... how to measure??
How to measure server as well...

Server-Client Link Latency
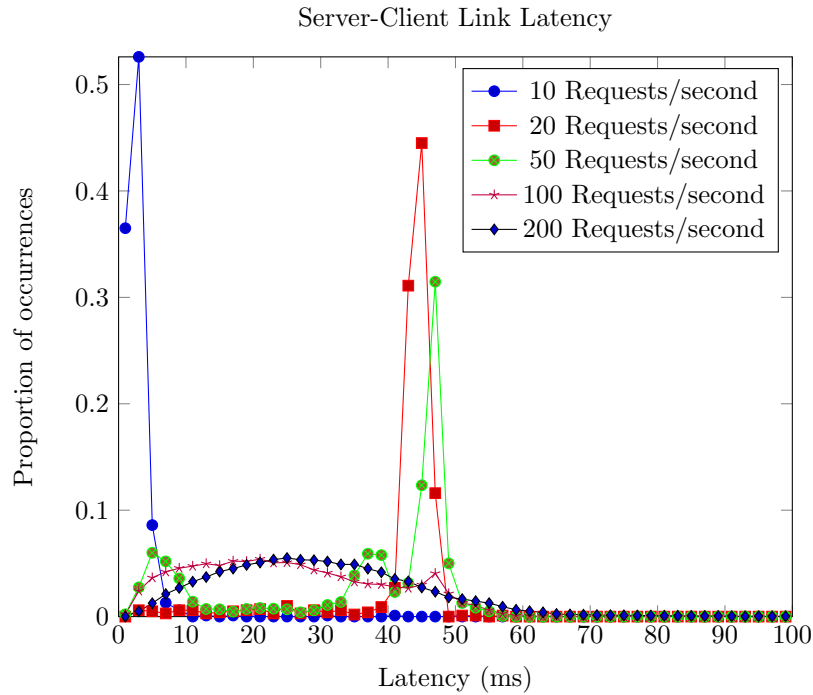
Figure 4.5: At very low load, the return latency is around 3ms. Until high load from multiple TCP connections, latencies are concentrated around 45ms. At high load a more normal distribution is achieved with a mean of 29ms.

Perhaps up to but not including sending op into network

Vector clocks:
CRDT: Network Cost of using vector clocks to ensure causal delivery. Discuss how we can't really do what ShareJS does since we don't know (1 packet in flight only) since we don't know when a packet has reached the final client in the network

Undo:
CRDT: Show cost of Undo with graph of no undo, CCI Undo, Immediate undo

4.3.4   ShareJS Performance

4.3.5   Core CRDT Performance

4.3.6   ShareJS vs CRDT for Text Editing

4.3.7   Network

Chapter 5

Conclusion

Bibliography

[1] Stephen E. Deering and Robert M. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. http://www.rfc-editor.org/rfc/rfc2460.txt. RFC Editor, Dec. 1998. URL: http://www.rfc-editor.org/rfc/rfc2460.txt.

[2] Dr. Andrew Moore. *Computer Networking*. 2016. URL: http://www.cl.cam.ac.uk/teaching/1516/CompNet/.

[3] Dr. R.J. Gibbens. *Computer Systems Modeling*. 2017. URL: http://www.cl.cam.ac.uk/teaching/1617/CompSysMod/.

[4] C A Ellis and S J Gibbs. "Concurrency Control in Groupware Systems". In: (1989).

[5] Colin J Fidge. "Timestamps in message-passing systems that preserve the partial ordering". In: (1987).

[6] Joseph Gentle. *ShareJS v0.6.3*. https://github.com/josephg/ShareJS/tree/0.6. 2013.

[7] Seth Gilbert and Nancy Lynch. "Brewer ' s Conjecture and the Feasibility of Consistent , Available , Partition-Tolerant Web Services". In: (2005), pp. 51–59.

[8] Santosh Kumawat and Ajay Khunteta. "Analysis of Operational Transformation Algorithms". In: *Proceedings of the International Conference on Recent Cognizance in Wireless Communication & Image Processing*. Springer. 2016, pp. 9–20.

[9] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[10] 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.

[11] 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.

[12] 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.

[13] Marc Shapiro and Nuno M. Preguiça. "Designing a commutative replicated data type". In: *CoRR* abs/0710.1784 (2007). URL: http://arxiv.org/abs/0710.1784.

[14] Marc Shapiro et al. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. INRIA, 2011, p. 50. URL: http://hal.inria.fr/inria-00555588.

[15] 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. DOI: `10.1080/01449298208914450`. URL: `http://dx.doi.org/10.1080/01449298208914450`.

[16] Mikito Takada. *Distributed Systems: for fun and profit.* 2013, pp. 1–62.

[17] *The Mother of All Demos, Reel 3.* `https://archive.org/details/XD300-25_68HighlightsAResearchCntAugHumanIntellect&start=286`. Accessed: 2017-03-01.

[18] Dr. Robert N. M. Watson. *Concurrent and Distributed Systems.* 2016. URL: `https://www.cl.cam.ac.uk/teaching/1617/ConcDisSys/`.

[19] 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.

[20] Stéphane Weiss, Pascal Urso and Pascal Molli. *Logoot: a P2P collaborative editing system.* Research Report RR-6713. INRIA, 2008, p. 13. URL: `https://hal.inria.fr/inria-00336191`.

[21] 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 [5], 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 $[c_1, c_2...c_n]$. c_p represents 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 §3.1.5 changed rule four from the previous section. For completeness, the full complete 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 §2.2.2. 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. We 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 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.

$\square$

Lemma 1. *A set of vectors $S = v_1, v_2...v_n$ 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, $\forall i.v^i_{result} = max(v^i_a, v^i_b)$.

The result of merging n vectors is $v^i_{result} = max(v^i_1, max(v^i_2, ...max(v^i_{n-1}, v^i_n)...))$, 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

C.1 Summary of Logs of Simple Experiment

```
---fully-connected, nonoptimized---
Total simulation duration: 848.547219206
Optimizations enabled: False
All clients converged to same result: True
Total insert events: 27
Total delete events: 0
Total insert packets sent: 125
Total size of insert packets sent: 11995
Average insert packet size (incl vector clock etc.): 95.96
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: 162
Expected number of packets sent - given optimal p2p network with everyone joining at s
Latency/wait time per client when requesting CRDT: [0, 416.77360960314616, 319.0786227
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: 68752
post-graph-init: 633464
post-clients-init: 651976
post-experiment: 1256448

---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 naiive broadcast in a p2p network: 18
Expected number of packets sent - given optimal p2p network with everyone joining at s
Latency/wait time per client when requesting CRDT: [0, 416.77360960314616, 319.0786227
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, nonoptimized---
Total simulation duration: 959.239037182
Optimizations enabled: False
All clients converged to same result: True
Total insert events: 27
Total delete events: 0
Total insert packets sent: 89
Total size of insert packets sent: 8489
Average insert packet size (incl vector clock etc.): 95.3820224719
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: 108
Expected number of packets sent - given optimal p2p network with everyone joining at s
Latency/wait time per client when requesting CRDT: [0, 416.77360960314616, 319.0786227
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: 71056
post-graph-init: 2482896
post-clients-init: 2491528
post-experiment: -2005448

---star, optimized---
Total simulation 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 :
Latency/wait time per client when requesting CRDT: [0, 416.77360960314616, 319.0786227
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 jo:
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 [**Technion**]. 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 that 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 [**sun1998operational**]. 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 [**preguica2009commutative**]. 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[**Shapiro2011**]. 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 [**ShareJS**].

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.