Joshua Send

Conflict Free Document Editing with Different Technologies

Computer Science Tripos – Part II

Trinity Hall

28th March 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 concurrent use of data in applications. 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 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 [14]. 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 [3] 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 Google Docs is so widely used might be 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 ends up with 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 [5]. However, if we forgo strong

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

consistency guarantees and settle for eventual consistency, we are able provide all three [17]. As we will see, achieving eventual consistency is non-trivial. The two prevailing approaches, 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, and run these experiments on an environment built around the open source library ShareJS (which implements OT) along with a comparative system I created based on a specific CRDT. 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 [section ref].

The custom CRDT on which the collaborative text editor is based is described in detail in the Implementation [section ref] section. 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. My approach was developed 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 an explanation of the required functionality provided by Martin Kleppmann. As a result, my solution is not optimal in all aspects, and could be improved upon in the future. It also falls into the class of 'tombstone' CRDTs, which mark elements as deleted rather than fully removing them, which forces the data structures to grow continuously over time. Other CRDTs are 'tombstone-free' and do not suffer from this inefficiency. Existing CRDTs of both types are discussed here.

1.3.1 Treedoc

Treedoc [10] 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. Each node in the tree contains one character, and the string contained in the buffer is retried using infix traversal. Each client has a copy of the same tree, and can insert new nodes 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. With the total order, all clients with copies of the tree will retrieve the same string from their Treedoc. Having a total order is an important property used to guarantee eventual consistency in CRDTs.

[perhaps insert Figure of large nodes/mininodes from the paper]

Deletes in this 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 authors propose an expensive commitment protocol to rebalance it. Not only is this inefficient, but also rather contrary to the spirit of CRDTs.

1.3.2 Logoot

Logoot [16] belongs to the class of text CRDTs which do not require tombstones for deletion. It achieves this by totally ordering identifiers, rather than relying on implicit causal dependencies between identifiers (which Treedoc embeds in the tree's branches). Logoot does generate identifiers using a tree, but each identifier contains the full path in the tree, which frees it of dependence on other nodes. This means that to delete, any client can simply remove the identifier and the data it tags.

Logoot also favors marking larger blocks of text with identifiers, rather than per-character. This, in combination with not needing tombstones, promises major efficiency gains over CRDTs such as Treedoc. Even further, two papers [9] [8] offer optimizations beyond the basic Logoot implementation by improving the strategy used to allocate new identifiers in the generator tree. However, these algorithms are specific to Logoot and of little relevance to this project.

Logoot is important as an example of a tombstone-free CRDT for text. Additionally, subsequent research enabled 'undo' and 'redo' functionality for this CRDT, 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, reversing an insert is not commutative with the original insertion. If it were, the removal of a nonexistent element, followed by its insertion would have to result in the same thing as insertion followed by removal. In the first case, the element is present, while in the second it is removed. These outcomes clearly are not the same.

Logoot Undo [15] 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 [REFERENCE NEEDED], this leads to some rather unexpected behavior. However, this approach is viable since increments and decrements commute and guarantee eventual convergence. In Logoot Undo, any client can undo any other client's operations which is called global undo. The use of a counter is identical to the undo mechanism I developed independently, though I chose to implement a local undo rather than global one, where clients can only undo their own operations.

Chapter 2

Preparation

2.1 Consistency Models

2.1.1 What is "Conflict Free"

One important question to answer is, what is the exact definition of conflict free. There appears to be more than one way of interpreting it. On one hand, there is the user's 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 or index in a text buffer. Conflict free then means that no data is lost, and after all operations are exchanged the resulting states agree.

The common conflicting operations in text editing are inserting characters into the same index of a shared text buffer, or simultaneously deleting the same characters. The second is easy to make conflict-free, and both the user and data oriented definitions of conflict agree – deleting a character concurrently or on a single user system should still result in the character disappearing. In the case of inserting text into the same index, the definitions cannot agree. Both users expect their own text to appear in the index they inserted at. However, in order to satisfy the data-centric definition we are not allowed to lose data, and must eventually present both users with the same string. The solution is to let one user 'win' and insert their characters at the desired index, and shift the other users' characters to appear after. Both operational transformations and CRDTs achieve this in fundamentally different ways.

[create figure]

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 [15].

- **Consistency:** All operations ordered by a precedence relation, such as Lamports happened-before relation [7], 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 we will see when discussing Undo in [reference needed].

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, and deleting the same character, and deleting a character while moving its position.

2.2.1 Operational Transformations

The easiest way to understand how operational transformations work is by example. The following three figures discuss each of the scenarios in turn.

2.2.2 Convergent Replicated Data Types

This section will provide an intuition for CRDTs in general, while the specific CRDT used for this project is outlined in chapter 3 [reference needed].

CRDTs, which were first formalized in a 2007 paper [11], 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, operations 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 TODO is a simple demonstration of how this works.

[Figure]

NEED to incorporate partial order of operations, commutativity of concurrent operations

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 [12]. 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[13]. However, either can be used to implement the other. This project uses an operation-based CRDT. Thus, the key property to fulfill is commutativity of operations.

If the communication layer requirements are met and commutativity is guaranteed, 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 [4] is an open source Javascript library implementing Operational Transformations which can be deployed on web browsers or NodeJS [1] clients. It is the core resource around which I built the comparative system to collect statistics from. To this end, it is useful to know more precisely how ShareJS operates and what kind of behavior might be expected. As are a large variety of algorithms that can enable OT [6], 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 versioned documents, an active server which orders and transforms operations, and primary supported actions 'insert' and 'delete'.

Replicated documents are versioned, and each operation applies to a specific version. The version number is used to transform operations against each other and detect concurrent changes. 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 delete operation:

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

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

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 global, 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 [EXPERIMENT NEEDED], which has an effect on memory consumption.

ShareJS clients can also only have one packet to be in flight to a server. This is why the operations above need to be combined into larger packets. However, this also has implications for packet size and quantity as network latency grows [EXPERIMENT NEEDED]. Additionally, since the server can reject operations that were generated locally at a given client, and have already been applied to the document, the clients must be able to undo operations, as well as transform any subsequent operations that have occurred against the inverse of the rejected one. So, the clients must each also have a list of past operations, which also affects memory use [EXPERIMENT NEEDED].

2.3 Analysis

2.3.1 Memory

This can be redone, for instance ops grow in size as document number grows

Given our rough understanding of CRDTs and ShareJS, we can make hypotheses about the quantitative results that might be obtained. In terms of basic memory requirements, each ShareJS client requires storing the current document string, along with a lot of past operations. At worst, each character in the string was delivered as an individual operation, so given n characters we expect O(2n) = O(n) memory usage. The server must also store a list of these operations, but the overall cost is still O(n). On the other hand, CRDTs at worst tag each character with a unique identifier. The largest identifier is O(log(n)) characters long (assuming increasing natural numbers as tags), which leads to O(nlog(n)) cost overall. This is slightly higher than ShareJS's linear growth.

2.3.2 Network

Given that the CRDT system will run over a P2P network, while ShareJS requires a server, as long as the P2P network is more connected than a star topology (equivalent to client/server), the average latency for all the clients to receive data should be lower. In fact, at best a P2P network will cut the time to receive an update to half of a client/server network.

In terms of number of characters sent over the network per action, it is difficult to make a prediction due to optimizations that may or may not be implemented. However, given a basic assumption of each character inserted into a CRDT also requiring an identifier to be sent, we get an O(nlog(n)) complexity again. On the other hand, ShareJS sends one character or word at a time, plus an index and a document version that the operation was performed on. If we have n characters in the document, we also have at most n versions ***?***. The length of the decimal string n is log(n) characters. Thus we get an approximate O(nlog(n)) packets sent for ShareJS.

2.3.3 Processor Load

The relative algorithmic simplicity of CRDTs versus OT hints that CRDTs should be computationally more efficient. If we assume that an insert and delete operation can be done in constant time in a CRDT, then the most expensive operation to be done is a linear time retrieval and update of the string displayed to the user. Operational transformations also need to update the displayed string, but also need to be transformed against any concurrent changes (either on clients or on the server). While this is hard to quantify, it is reasonable to expect OT to be more processor intensive than CRDTs.

2.3.4 Client-Server versus Peer to Peer

It is worth examining what other reasons there might be for using a system that is capable of running over a P2P network. Generally, a key element is privacy. A P2P network can run over a secure, anonymous network such as Tor [2] and since no middleware needs to intercept and read packets, encryption may be used. OT systems almost always require a server, which may need to transform operations against each other which requires transmitting all operations in plain text and kills any hope of privacy. One benefit of using a central server is that there is a natural cloud repository in which to store the contents of documents; a P2P network either requires some peers to be connected in order to download the latest version, or a server to have a repository of documents. Similar issues face state replay for new clients that join an active network; this is examined in section [SECTION REF]. Luckily, in terms of privacy, a central repository for CRDT based documents would not need to be able to read the contents, just distribute them on demand. Lastly, established P2P networks have further benefits such as lacking single points of failure, lower probability of downtime and lower operational cost to the provider, but these properties and their implications are not in the scope of this project.

[2] `https://www.torproject.org/docs/faq`

Table 2.1: Project Goals, Priority, and 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 ShareJS system capable of running experim	High	Medium
Write log analysis scripts	Medium	Low

2.4 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. Secondly, the Part IB course on Concurrent and Distributed Systems [4] provided valuable background towards Lamport and Vector clocks, causality, and total orderings. Lastly, the Mobile and Sensor Systems course [5] gave me some ideas when seeking alternatives to the flooding implemented in my network simulation.

2.5 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 comparing ShareJS and the CRDT based system

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

[3] `http://www.cl.cam.ac.uk/teaching/1617/CompSysMod/`
[4] `https://www.cl.cam.ac.uk/teaching/1617/ConcDisSys/`
[5] `http://www.cl.cam.ac.uk/teaching/1617/MobSensSys/`

2.6 Software Engineering

2.6.1 Libraries

ShareJS [4] 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 [6] package manager. The other large library I used was D3.js [7], 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.

The full list of package dependencies required directly and indirectly can been found in Appendix ***TODO***.

2.6.2 Languages

The three main implementation languages, by lines of code, are Typescript [8], Python 2.7 [9], 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 platform.

2.6.3 Tooling

The aforementioned testing platform has to be a web browser for compatibility with ShareJS. The most developer friendly choices are Mozilla Firefox [10] and Google Chrome [11], as both come with sophisticated debuggers and script inspection capabilities. However, both have issues for this project. Firstly, measuring memory consumption in Firefox is difficult, and the relatively hidden API that enables it is complex and badly documented [12]. On the other hand, Chrome offers a simple interface to measure memory when certain

[6] https://www.npmjs.com/
[7] https://d3js.org/
[8] http://www.typescriptlang.org/
[9] https://www.python.org/
[10] https://www.mozilla.org/en-US/firefox/new/
[11] https://www.google.com/chrome/
[12] https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Reference/Interface/nsIMemoryReporterManager

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 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.

2.6.4 Backup Strategy and Development Machine

Backups and data safety were mentioned in the project proposal. Github [13] provided the primary backup, with commits at important checkpoints and at least once per work day. The local repository also lives in my Dropbox folder for continuous cloud backups. To prevent data loss in event of system failure, user data resides on its own hard drive, separate from the primary development operating system Ubuntu 14.04 LTS x64. The MCS computers are the alternative development machines in case of complete failure or loss of laptop.

2.7 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.

2.7.1 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 focal point of this project, I did my best to keep everything simple. My system assumes the network guarantees in order delivery, the network does not during simulation, and is capable of a broadcast to all peers of a node. We will see how to relax some of these guarantees

[13] `github.com`

in section [section ref]. Broadcast is not typically found in Internet applications IPv6 does not even include any 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 $O(n^2)$ packets, where n is the number of clients in a fully connected network. This property can be seen in the Evaluation [section ref] section [EXPERIMENT NEEDED]. However, though it has downsides, broadcast is simple to simulate 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 fairly important. My initial focus was on a fully connected P2P topology to contrast with the clients/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.

2.7.2 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 Javascript JSON object. The second is to log network packets on the application layer. That is, rather than intercepting and logging packet information at the operating system, I log data about the 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, has access to an editable text area, and a simplified network stack. The network stack that each client has access to 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.

The system architecture is diagrammatically shown in Figure [TODO].

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 process of executing these steps is described in the following subsection. This is followed by a description of the network simulation and the design choices made within it. ***ugly wording***

[1] `https://webrtc.org`

3.1.2 CRDT

To briefly review section [section ref], 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 some optimizations I added.

Structure and Functionality

In the Related Work section [section ref] 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, a pointer, and is associated with a unique identifier.

The 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

Inserting characters is done as in standard linked lists – find the node after which the insert should occur, rewrite its pointer to point to the new node, and repair the broken link with the new node's pointer.

Figure of Linked List "insert" in my context

Deletes are handled by adding 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 we might want to do on a linked list are O(n): finding a node to insert after or delete requires at worst a scan of the whole list. A read is BigTheta(n) ***Todo***. A much better approach is to implement the linked list within a hash table. Since character identifiers are required to be unique, we can use them as keys in our map and achieve O(1) lookup times for any node. Figure [todo] gives a full CRDT using this structure. Hashing means insert and delete operations can complete in constant time, and only read takes BigTheta(n) to retrieve the document.

[Figure. Caption describe key as identifier and value as {char: 'h', 'next': etc...}]

Javascript provides two native objects capable of mapping. One is the Map [2] structure, and another is the standard Javascript object (referred to as Objects from now on). Both have advantages and disadvantages: Objects only allow strings and numbers to be used as keys, while Maps can use arbitrary entities. On the other hand, Objects serialize very easily to JSON (Javascript Object Notation), while Maps would require their own conversion functions. As we will see in the next section, CRDT keys are pairs of numbers. Thus, the sensible structure would be a Map, as we can map from pairs to values. Unfortunately, Maps natively do not hash the contents of the keys, but only the reference pointing to the key. I implemented both Map and Object variants of the CRDT, but the original pointer to the identifier used is not retained; it is not possible to retrieve the value associated with a given identifier. To solve this issue I serialized the key pair into a string on each lookup and insert (which, being immutable in Javascript are compared by content rather than pointer). At that point Maps have no more advantages over Objects and a distinct disadvantage in terms of serializing to JSON. Thus, I eventually settled on CRDTs implemented using Objects as lookup structures.

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 character 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.

The other part of character identifiers is drawn from the Natural numbers [insert Fancy N]. Every client maintains a Lamport [7] clock that is incremented on each local insert and updated on receipt of an insert operation from another client. To create an identifier we then use the following definition.

Define an identifier generated by client i to be a pair (t, *cid*) where t is the next value Ti, the value of the Lamport clock on client I. *cid* is the globally unique identifier of client I.

Since each client provides a fresh t per character, and *cid* is globally unique, each pair generated in the system is guaranteed to be unique.

We can now define a total order over the identifiers

[todo, formal total order definition]

[describe why we even need Lamport clock... I think this has to do with the fact that if one client does loads and loads of edits then it always wins the concurrent insert...? This

[2] `https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Map`

is pretty arbitrary and could probably be done without lamport anyway... need to think about it a bit more]

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 c 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

```
interface InsertMessage {
    id: string,
    char: string,
    after: string
}
```

Listing 3.1: Insert Bundle Type Signature

Incorporating an insert bundle into the CRDT is the following sequence of steps

1. Locate in the hash table the node $Prev$ corresponding to the $B.after$ identifier received
2. Insert into the table $B.id$ mapped to a new node N containing the character $B.c$ and a copy of $Prev$'s $next$ pointer
3. Rewrite $Prev$'s $next$ pointer to point to$N$

This is the standard procedure to insert a new node into a 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 $deleteId$

```
interface DeleteMessage {
    deleteId: string
}
```

Listing 3.2: 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 section [section ref], 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 – 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 [10]. In effect, the system periodically executes a distributed garbage collection. While possible, I have not implemented this protocol. While remove tombstones may be necessary after the data structure becomes too large, until such a point tombstones are useful in implementing undo functionality for the text editor. This is discussed in section [section ref].

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)'s *after* pointer and each new node points to *t + (i+1), c)*. The final node points to the original target of (t_{after}, c).

This insert optimization has potentially massive gains in terms of network efficiency. At best, an entire document could be inserted at once. A more likely scenario is word-by-word

or line-by-line insertion. At worst, the number of bundles generated is still linear. Thus we conclude the resulting complexity is somewhere between constant and linear.

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. However, this is strictly a constant multiplier gain.

3.1.5 Network simulation

The section describes the network 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.

Assumptions

The CRDT I implemented requires that messages be delivered causally [lecture notes - http://www.cl.cam.ac.uk/teaching/1617/ConcDisSys/2017-DistributedSystems-1B-L4-handout.pdf]. We define the happens-before relation $a \rightarrow b$ to be true whenever a happens before b on the same process, for example

$$Receive\ Insert\ "Hello" \rightarrow Insert\ "World"$$

We then require that all events ordered by $\rightarrow$ be delivered in a valid ordering according to $\rightarrow$. We call this "Causal Order". This defines a partial order, since concurrent operations A and B are valid and are neither $A \rightarrow B$ nor $A \rightarrow B$ holds. Concurrent operations can be delivered in any order since these operations are guaranteed to commute by the properties of the CRDT.

The CRDT requires causal order delivery since operations depends on each other. It is not valid to insert a link into a linked list after a node that does not exist: the links in the CRDT embed the causal ordering.

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, I also wrote my simulation such that a client that receives a packet from the network immediately floods it to its peers before generating and broadcasting dependent operations. Along with the assumptions above, the network provably guarantees causal delivery

Guarantee of Causal Delivery. Proof by contradiction. Assume a client receives some packets A and B such that $A \not\to B$ and that at some prior point, it must have been the case that $A \to B$. This implies that somewhere in the network A and B switched order. However, on any individual link, A and B stay in order by network assumption 2 above. At any node, packets cannot switch order either, since the implementation immediately forwards an incoming packet to its neighbors (If a particular client generated B, it is put on a link after A by the same condition). Lastly, because the network does not change topology and the protocol is deterministic flooding, packets do not dynamically adjust routes and so any packet that begins in order A, then B stays in order A, B. Thus, in no case can A and B switch order and our assumption must be incorrect. Either $A \to B$ or it must have been the case that $A \not\to B$ to begin with. The first case proves our goal and the second is irrelevant. □

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

Abstraction

My network simulation is implemented in two core modules: a Network 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 Network Manager abstracts away the bottom layers.

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

```
interface NetworkInterface {
  isEnabled: () => boolean;
  enable: () => void;
  setClientId: (ClientId) => void;
  setManager: (NetworkManager) => void;
  requestCRDT: (ClientId) => void;
  returnCRDT: (ClientId, MapCRDTStore) => void;
  broadcast: (PreparedPacket) => void;
  receive: (NetworkPacket) => void;
}
```

Listing 3.3: NetworkInterface Type Signature (cleaned)

The primary mechanism for disseminating a packet to other clients is via the `NetworkInterface.broadcast` method, which in turn calls the `broadcast` method of the `NetworkManager` (see [section ref]). It accepts a `PreparedPacket` which is an object that contains a bundle (such as `InsertMessage` or `DeleteMessage` from section [section ref]) and a tag which the receiver uses to disambiguate the type of the incoming packet. This is required since packets are serialized to strings when sent over the network and all type information is lost in the process.

```
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 }
```

Other types of bundles that can be sent are `RequestCRDTMessage` and `ReturnCRDTMessage`. These are 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 During the execution of a simulation, clients may join the network (but not leave) at any time. Only the first client gets to create a fresh CRDT. Any other client must request a copy of that CRDT when joining. This means that there is at least 1 round trip time before a new editor can come online. The packets which deliver the CRDT can become quite large – the later a client joins, the larger the packet. However, they would also have avoided many individual packets being delivered to them in the meantime. There is a trade off here as well. A new client could begin with a empty CRDT and obtain 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 sort of server store them), and the local client would have to spend computational time reintegrating all the changes. The downside to obtaining an up to date copy is that there is no straightforward way to do a partial replay. If a client simply has an out of date CRDT, it must request an entire new copy.

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 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, which is 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 3.4: The Scheduler.addEvent method

My scheduler is somewhat more sophisticated than might be expected in that it takes two keys for scheduling: a primary key *time*, and a secondary key *clock*. The need for this arose when submitting multiple packets from a single client at the same time - the original underlying data structure, a heap, makes no first-in first-out assurances. Thus, packets on a link could arrive out of order, which violated one of the guarantees the network has to provide [reference needed]. To fix this, my scheduler breaks ties using *clock*, which is usually a monotonically increasing counter provided by the caller.

The key property of a correct scheduler, as noted in the Part II Computer Systems Modeling [2, slide 120] course, is that the next executed event be the one with the least remaining time. To do this efficiently, I implemented a heap that orders elements based on the two keys discussed above. Using a heap, we get $O(log(n))$ 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 continues this until paused or the event queue is empty.

Because the simulation needs to be seeded with events and simulated action in order to do anything useful, before letting the execution begin, the scheduler is also used to add a set of mock insert and delete events, which together constitute an experiment. This is discussed in more detail in [section ref]. Once running, its primary use is delivering packets.

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 known.

The alternative 'time' that can be used is real time. The amount of time until some next event E is given in milliseconds, rather than a Δt which is skipped over once executed. 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. To handle this difficulty, the scheduler runs, in order, all ready events whenever it wakes up rather than just one at a time.

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 lets it. 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.

Luckily, the timer-driven scheduler is useful in the comparative system. In it, packets are actually delivered via sockets and the operating system, which means that logical time can no longer be used. This is discussed further in section [section ref].

Causal Delivery

Until this point, the network has guaranteed causal delivery of packets based on very strong assumptions and knowledge of the implementation of the system [reference needed to proof]. We can relax these to allow out of order delivery and a changing network topology and remove need for a specific implementation. This can be done by ensuring ensures causal delivery A commonly used method for this is vector clocks.

3.2 ShareJS Comparative Environment

3.2.1 Overview

3.3 Experiments and Automated Log Analysis

3.3.1 Separation of Concerns

3.3.2 Experiment Design

3.3.3 Log Analysis

3.4 Extension: Local Undo

Chapter 4

Evaluation

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 R.J. Gibbens. *Computer Systems Modeling Lecture Notes*. 2017. URL: `http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm`.

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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`.

[12] 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`.

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

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

[15] 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.

[16] 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.

[17] 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

Latex source

A.1 diss.tex

```
\documentclass[12pt,a4paper,twoside,openright]{report}


\usepackage[pdfborder={0 0 0}]{hyperref}    % turns references into hyperlinks
\usepackage[margin=25mm]{geometry}  % adjusts page layout
\usepackage{graphicx}  % allows inclusion of PDF, PNG and JPG images
\usepackage{verbatim}
\usepackage{docmute}   % only needed to allow inclusion of proposal.tex
\usepackage{url}
\usepackage[parfill]{parskip}
\usepackage{booktabs}
\usepackage{fixltx2e}
\usepackage{amsmath}
\usepackage{amsthm}
\usepackage{listings}


\usepackage{fancyvrb,newverbs,xcolor}


\usepackage[UKenglish]{babel}% Recommended
\usepackage[bibstyle=numeric,citestyle=numeric,backend=biber,natbib=true]{biblatex}

\addbibresource{refs.bib}% Syntax for version >= 1.2


\raggedbottom                           % try to avoid widows and orphans
\sloppy
\clubpenalty1000%
\widowpenalty1000%

\renewcommand{\baselinestretch}{1.1}    % adjust line spacing to make
                                        % more readable



\lstdefinelanguage{Typescript}{
keywords={break, case, catch, continue, debugger, default, delete, do, else, finally, for, function, if, in, instanceof, new
keywordstyle=\color{blue}\bfseries,
ndkeywords={interface, class, extends, export, void, number, boolean, throw, implements, import, this},
ndkeywordstyle=\color{orange}\bfseries,
identifierstyle=\color{black},
sensitive=false,
```

```
comment=[l]{//},
morecomment=[s]{/*}{*/},
commentstyle=\color{purple}\ttfamily,
stringstyle=\color{red}\ttfamily,
morestring=[b]',
morestring=[b]"
}

\lstset{
language=Typescript,
backgroundcolor=\color{lightgray},
extendedchars=true,
basicstyle=\footnotesize\ttfamily,
showstringspaces=false,
showspaces=false,
numbers=left,
numberstyle=\footnotesize,
numbersep=9pt,
tabsize=2,
breaklines=true,
showtabs=false,
captionpos=b
}


\definecolor{cverbbg}{gray}{0.93}
\newenvironment{cverbatim}
 {\SaveVerbatim{cverb}}
 {\endSaveVerbatim
  \flushleft\fboxrule=0pt\fboxsep=.5em
  \colorbox{cverbbg}{\BUseVerbatim{cverb}}%
  \endflushleft
}
\newenvironment{lcverbatim}
 {\SaveVerbatim{cverb}}
 {\endSaveVerbatim
  \flushleft\fboxrule=0pt\fboxsep=.5em
  \colorbox{cverbbg}{%
    \makebox[\dimexpr\linewidth-2\fboxsep][l]{\BUseVerbatim{cverb}}%
  }
  \endflushleft
}
\newcommand{\ctexttt}[1]{\colorbox{cverbbg}{\texttt{#1}}}
\newverbcommand{\cverb}
  {\setbox\verbbox\hbox\bgroup}
  {\egroup\colorbox{cverbbg}{\box\verbbox}}



\begin{document}




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title


\pagestyle{empty}

\rightline{\LARGE \textbf{Joshua Send}}

\vspace*{60mm}
\begin{center}
\Huge
\textbf{Conflict Free Document Editing with Different Technologies} \\[5mm]
Computer Science Tripos -- Part II \\[5mm]
```

```
Trinity Hall \\[5mm]
\today  % today's date
\end{center}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Proforma, table of contents and list of figures

\pagestyle{plain}

\chapter*{Proforma}

{\large
\begin{tabular}{ll}
Name:               & \bf Joshua Send                       \\
College:            & \bf Trinity Hall                      \\
Project Title:      & \bf Conflict Free Document Editing with Different Technologies \\
Examination:        & \bf Computer Science Tripos -- Part II, June 2017  \\
Word Count:         & \bf 1587\footnotemark[1] \\
Project Originator: & Joshua Send                    \\
Supervisor:         & Stephan Kollmann                    \\
\end{tabular}
}
\footnotetext[1]{This word count was computed
by \texttt{detex diss.tex | tr -cd '0-9A-Za-z $\tt\backslash$n' | wc -w}
}
\stepcounter{footnote}


\section*{Original Aims of the Project}


TODO\footnote{A normal footnote without the
complication of being in a table.}


\section*{Work Completed}

TODO

\section*{Special Difficulties}

TODO

\newpage
\section*{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.

\bigskip
\leftline{Signed TODO [signature]}

\medskip
\leftline{Date TODO [date]}

\tableofcontents

\listoffigures

\newpage
\section*{Acknowledgements}

TODO
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% now for the chapters

\pagestyle{headings}

\chapter{Introduction}

Real time interaction between users is becoming an increasingly important feature to many applications, from word processing


\section{Motivation}

Realtime collaborative editing was first motivated by a demonstration in the Mother of All Demos by Douglas Engelbart in 196

The convergence, or consistency, property above is the hardest to provide -- it is easy to create a system where the last wr


\section{Overview}
This project aims to examine the trade-offs made when implementing highly distributed and concurrent document editing with O

The custom CRDT on which the collaborative text editor is based is described in detail in the Implementation [section ref] s

One extension, adding undo functionality to the CRDT, was also completed. My approach was developed originally, before readi

\section{Related Work}

Part of the challenge of this project was to develop my CRDT and associated algorithms based only on an explanation of the r

\subsection{Treedoc}

Treedoc \cite{preguica2009} is a replicated text buffer; an ordered set that supports insert-at and delete operations. This

[perhaps insert Figure of large nodes/mininodes from the paper]

Deletes in this Treedoc are handled by marking a node as deleted (but the node remains in the structure). Thus Treedoc falls

\subsection{Logoot}

Logoot \cite{weiss2008} belongs to the class of text CRDTs which do not require tombstones for deletion. It achieves this by

Logoot also favors marking larger blocks of text with identifiers, rather than per-character. This, in combination with not

Logoot is important as an example of a tombstone-free CRDT for text. Additionally, subsequent research enabled 'undo' and 'r


\subsection{Logoot-Undo}

CRDTs generally struggle to provide an undo mechanism since the concept of reversing an update to the data structure is fund

Logoot Undo \cite{weiss2010undo} proposes to resolve this by essentially tagging each identifier with a 'visible' counter. A


\chapter{Preparation}


\section{Consistency Models}

\subsection{What is "Conflict Free"}

One important question to answer is, what is the exact definition of conflict free. There appears to be more than one way of

The common conflicting operations in text editing are inserting characters into the same index of a shared text buffer, or s

[create figure]

\subsection{CCI Consistency Model}
```

```
The commonly used consistency model for concurrent document editing is the CCI model. The definition here is borrowed from \

\begin{itemize}
\item \textbf{Consistency:} All operations ordered by a precedence relation, such as Lamports happened-before relation \cite

\item \textbf{Convergence:} The system converges if all replicas are identical when the system is idle.

\item \textbf{Intention Preservation:} The expected effect of an operation should be observed on all replicas. This is commo

\begin{itemize}
\item \textit{delete}  A deleted line must not appear in the document unless the deletion is undone.

\item \textit{insert}  A line inserted on a peer must appear on every peer; the order relation between the document lines an

\item  \textit{undo}  Undoing a modification makes the system return to the state it would have reached if this modification

\end{itemize}

\end{itemize}

The given definition of intention preservation is accepted, but may produce some unexpected results as we will see when disc


\section{Achieving Eventual Consistency}

As mentioned briefly in the prior section, operational transformations and CRDTs aim to achieve eventual convergence on all

\subsection{Operational Transformations}

The easiest way to understand how operational transformations work is by example. The following three figures discuss each o




\subsection{Convergent Replicated Data Types}

This section will provide an intuition for CRDTs in general, while the specific CRDT used for this project is outlined in ch

CRDTs, which were first formalized in a 2007 paper \cite{shapiro2007}, trade the complex algorithms used in OT for a more co

[Figure]

***NEED to incorporate partial order of operations, commutativity of concurrent operations***

There are technically two classes of CRDTs: state- and operation-based. State-based CRDTs disseminate the entire local state

If the communication layer requirements are met and commutativity is guaranteed, all clients will to converge to an identica

\subsection{ShareJS}

ShareJS \cite{sharejs} is an open source Javascript library implementing Operational Transformations which can be deployed o

Replicated documents are versioned, and each operation applies to a specific version. The version number is used to transfor

An Insert operation for adding text at index 100 in document version 1:
\begin{lcverbatim}
{v:1, op:[{i:'Hello World', p:100}]}
\end{lcverbatim}

A delete operation:
\begin{lcverbatim}
{v:1, op:[{d:'Hello', p:100}]}
\end{lcverbatim}

Multiple operations may be sent in one packet:
\begin{lcverbatim}
```

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

\vspace{5mm}

The library contains both client and a server code. The server provides a global, serialized order of operations to be appli

ShareJS clients can also only have one packet to be in flight to a server. This is why the operations above need to be combi


\section{Analysis}

\subsection{Memory}

***This can be redone, for instance ops grow in size as document number grows***

Given our rough understanding of CRDTs and ShareJS, we can make hypotheses about the quantitative results that might be obta

\subsection{Network}

Given that the CRDT system will run over a P2P network, while ShareJS requires a server, as long as the P2P network is more

In terms of number of characters sent over the network per action, it is difficult to make a prediction due to optimizations

\subsection{Processor Load}

The relative algorithmic simplicity of CRDTs versus OT hints that CRDTs should be computationally more efficient. If we assu


\subsection{Client-Server versus Peer to Peer}

It is worth examining what other reasons there might be for using a system that is capable of running over a P2P network. Ge

\section{Starting Point}

As stated in the proposal, I had prior experience with ShareJS, which was leveraged when creating the comparative system. Ad

As the project progressed, several courses contributed or reaffirmed ideas I could use. Notably, the Computer Systems Modeli

\section{Requirements Analysis}
To reiterate the success criteria listed in the project proposal, I hoped to

\begin{enumerate}
\item Implement a concurrent, distributed text editor based on CRDTs
\item Pass correctness tests for this CRDT
\item Obtain and compare quantitative results comparing ShareJS and the CRDT based system
\end{enumerate}

Points one and three have multiple unspecified subgoals. For clarity, Table lists all of these and their respective importan

\begin{center}
\begin{table}
\centering
\caption{Project Goals, Priority, and Difficulty}
\label{my-label}
\begin{tabular}{lll}
\toprule
Implement and unit test core CRDT               & High   & Medium \\ \midrule
Implement network simulation                    & High   & High   \\ \midrule
Optimize CRDT Insert                            & Low    & Low    \\ \midrule
Design experiment format                        & High   & Low    \\ \midrule
Create ShareJS system capable of running experim & High   & Medium \\ \midrule
Write log analysis scripts                      & Medium & Low    \\ \bottomrule
\end{tabular}
\end{table}
\end{center}
```

```
\section{Software Engineering}

\subsection{Libraries}
ShareJS \cite{sharejs} is the main external resource I required. It is released under the MIT license. I used the simpler Sh

The full list of package dependencies required directly and indirectly can been found in Appendix ***TODO***.

\subsection{Languages}
The three main implementation languages, by lines of code, are Typescript \footnote{\url{http://www.typescriptlang.org/}}, P

\subsection{Tooling}
The aforementioned testing platform has to be a web browser for compatibility with ShareJS. The most developer friendly choi

Before starting this project, I was already familiar with a specific Typescript development stack and environment. The wide

\subsection{Backup Strategy and Development Machine}
Backups and data safety were mentioned in the project proposal. Github \footnote{\url{github.com}} provided the primary back


\section{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 flexib

\subsection{Network Simulation}
One broad category of decisions has to do with the network simulation I implemented. Because I had no experience with simula

While the broadcast is a useful simplification, the topology of a P2P network affects a system's functionality nearly as str

%To aid debugging and visualization [GB?] I also decided to build a dynamic graphical network representation that could be r

\subsection{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 num


\chapter{Implementation}

The chapter describes the implementation of both real time editing systems, firstly the one based on CRDTs and secondly the


\section{CRDT-based system}

\subsection{Overview}
The high level components that make up my CRDT-based text editor are the user interface, the CRDT, and the network simulatio

The system architecture is diagrammatically shown in Figure [TODO].

Upon interaction with a client's text interface, the CRDT is modified and the operations generated are passed to the network


\subsection{CRDT}
To briefly review section [section ref], a text CRDT can be thought of as a set containing characters tagged with totally or

This subsection goes through the structure and capabilities of the CRDT I utilized, how character identifiers are generated

\subsubsection{Structure and Functionality}
In the Related Work section [section ref] various structures were mentioned, such as Treedoc which stores characters in the

The CRDT needs to implement three core methods in order to support text editing
\begin{enumerate}
\item \textbf{Insert:} Add a character at a specific index or location
\item \textbf{Delete:} Remove or mark as deleted any given character
\item \textbf{Read:} Retrieve the characters in order and return them as a string
\end{enumerate}

Inserting characters is done as in standard linked lists -- find the node after which the insert should occur, rewrite its p
```

```
Figure of Linked List "insert" in my context

Deletes are handled by adding to the relevant link a 'deleted' tag. Lastly, the read operation is a linear time traversal ov

Various data structures were considered when implementing this linked list. The most intuitive approach is to implement each

[Figure. Caption describe key as identifier and value as \{char: 'h', 'next': etc...\}]

Javascript provides two native objects capable of mapping. One is the Map \footnote{\url{https://developer.mozilla.org/en/do

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

\subsubsection{Identifiers}

Recall that CRDT identifiers are required to be globally unique and totally ordered. My CRDT is advantageous over tree-based

The other part of character identifiers is drawn from the Natural numbers [insert Fancy N]. Every client maintains a Lamport

Define an identifier generated by client i to be a pair (\textit{t}, \textit{cid}) where \textit{t} is the next value Ti, th

Since each client provides a fresh \textit{t} per character, and \textit{cid} is globally unique, each pair generated in the

We can now define a total order over the identifiers

[todo, formal total order definition]

[describe why we even need Lamport clock... I think this has to do with the fact that if one client does loads and loads of


\subsubsection{Operations}
The text CRDT supports two core modifying operations: insert and delete.

\paragraph{Insert}
Inserting a character into the text document generates an insert operation. An insert operation is stored and transmitted as

An insert bundle $B$ contains
\begin{itemize}
\item A unique identifier $id$ for the character
\item The character $c$ itself
\item The node after which to insert the character denoted $after$. This corresponds to the position in the linked list at w
\end{itemize}

\vspace{3mm}
\begin{lstlisting}[caption=Insert Bundle Type Signature]
interface InsertMessage {
    id: string,
    char: string,
    after: string
}
\end{lstlisting}

Incorporating an insert bundle into the CRDT is the following sequence of steps
\begin{enumerate}
\item Locate in the hash table the node $Prev$ corresponding to the $B.after$ identifier received
\item Insert into the table $B.id$ mapped to a new node $N$ containing the character $B.c$ and a copy of $Prev$'s $next$ poi
\item Rewrite $Prev$'s $next$ pointer to point to$N$
\end{enumerate}
This is the standard procedure to insert a new node into a linked list.

\paragraph{Delete}
Deleting a character from the text document generates a delete operation, which is transmitted as a bundle $B$ containing
\begin{itemize}
\item The target character's identifier to be deleted $deleteId$
\end{itemize}

\vspace{3mm}
\begin{lstlisting}[caption=Delete Bundle Type Signature]
```

```
interface DeleteMessage {
    deleteId: string
}
\end{lstlisting}

Incorporating a delete bundle into the CRDT is straightforward
\begin{enumerate}
\item Locate the node $N$ corresponding to $B.deleteId$
\item Set a boolean flag in $N.d$ to true
\end{enumerate}

\subsection{Tombstones}
The delete operation described previously never removes nodes, but leaves them behind as tombstones. Some CRDTs, such as Log

The process of establishing that each client has received and executed an operation can be achieved using an expensive commi

\subsection{Optimizations}
The insert operation produces a bundle which contains exactly one character, one identifier, and one \textit{after} tag. We

An optimized insert bundle contains
\begin{itemize}
\item A unique identifier  \textit{(t, cid)} for the first character
\item The character sequence \textit{s} itself
\item The node after which to insert the character sequence denoted \textit{(t\textsubscript{after}, c)}
\end{itemize}
The receiver CRDT incorporates the bundle by generating a new node for each character \textit{s\textsubscript{i}} with ident

This insert optimization has potentially massive gains in terms of network efficiency. At best, an entire document could be

Another optimization I added was renaming tags and names to be as short as possible (often single characters), so that the r

\subsection{Network simulation}
The section describes the network that delivers operation bundles from one client to another. First, I will detail the initi

\subsubsection{Assumptions}
The CRDT I implemented requires that messages be delivered causally [lecture notes - http://www.cl.cam.ac.uk/teaching/1617/C
We then require that all events ordered by $\rightarrow$ be delivered in a valid ordering according to $\rightarrow$. We cal
This defines a partial order, since concurrent operations $A$ and $B$ are valid and are neither $A \rightarrow B$ nor $A \ri

The CRDT requires causal order delivery since operations depends on each other. It is not valid to insert a link into a link

\underline{Network Assumptions}. These are guaranteed by the simulation.
\begin{enumerate}
\item Unchanging network topology
\item In order delivery on any single link in the network
\item No packets are lost
\end{enumerate}

Along with this, I also wrote my simulation such that a client that receives a packet from the network immediately floods it

\begin{proof}[Guarantee of Causal Delivery]
Proof by contradiction. Assume a client receives some packets $A$ and $B$ such that $A \not\rightarrow B$ and that at some p
\end{proof}

That the network is able to guarantee causal delivery is a very strong assumption and cannot be made in general. We will see

\subsubsection{Abstraction}
My network simulation is implemented in two core modules: a Network Manager which is shared between all simulated clients, a

\paragraph{Network Interface} The essential parts of the Network Interface Typescript signature are shown below.

\begin{lstlisting}[caption=NetworkInterface Type Signature (cleaned)]
interface NetworkInterface {
isEnabled: () => boolean;
enable: () => void;
setClientId: (ClientId) => void;
setManager: (NetworkManager) => void;
```

```
requestCRDT: (ClientId) => void;
returnCRDT: (ClientId, MapCRDTStore) => void;
broadcast: (PreparedPacket) => void;
receive: (NetworkPacket) => void;
}
\end{lstlisting}

The primary mechanism for disseminating a packet to other clients is via the \lstinline|NetworkInterface.broadcast| method,
\begin{lstlisting}
interface PreparedPacket {
type: "i" | "d" | "reqCRDT" | "retCRDT",    // insert or delete message, or request CRDT or return CRDT
bundle: CRDTTypes.InsertMessage | CRDTTypes.DeleteMessage | RequestCRDTMessage | ReturnCRDTMessage;
}
\end{lstlisting}

Other types of bundles that can be sent are \lstinline|RequestCRDTMessage| and \lstinline|ReturnCRDTMessage|. These are spec

\paragraph{Joining the Network}
During the execution of a simulation, clients may join the network (but not leave) at any time. Only the first client gets t



\paragraph{Network Manager}
The lower layers of the network stack are provided by the Network Manager. It has two key methods: \lstinline|NetworkManager


\subsubsection{Scheduler}
Although this subsection falls under the Network Simulation section of this document, the scheduler is an altogether more ge

A simulation scheduler is responsible for mutating system state, based on events given to it to be executed at specific time
\begin{lstlisting}[caption=The Scheduler.addEvent method]
public addEvent(time: number, clock: number, action: any) {
    let heapElem: DualKeyHeapElement = {
        pKey: time,
        sKey: clock,
        payload: action
    };
    this.heap.insert(heapElem);
}
\end{lstlisting}

My scheduler is somewhat more sophisticated than might be expected in that it takes two keys for scheduling: a primary key $

The key property of a correct scheduler, as noted in the Part II Computer Systems Modeling \cite[slide 120]{compsysmodeling}

When the simulation is running, the scheduler removes the top event $E$ off the heap, decreases all remaining events' primar

Because the simulation needs to be seeded with events and simulated action in order to do anything useful, before letting th

\paragraph{Time}
The concept of time in a simulation is generally taken to be ``logical time''. The system begins in $t = 0$, and each subseq

The alternative 'time' that can be used is real time. The amount of time until some next event $E$ is given in milliseconds,

In this project, I implemented both an event-driven scheduler and a timer-driven scheduler. The timer-driven version is usef

Luckily, the timer-driven scheduler is useful in the comparative system. In it, packets are actually delivered via sockets a


\subsubsection{Causal Delivery}
Until this point, the network has guaranteed causal delivery of packets based on very strong assumptions and knowledge of th
A commonly used method for this is vector clocks.

\section{ShareJS Comparative Environment}

\subsection{Overview}
```

```
\section{Experiments and Automated Log Analysis}

\subsection{Separation of Concerns}

\subsection{Experiment Design}

\subsection{Log Analysis}

\section{Extension: Local Undo}


\chapter{Evaluation}


\chapter{Conclusion}



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% the bibliography
\addcontentsline{toc}{chapter}{Bibliography}
\printbibliography

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% the appendices
\appendix

\chapter{Latex source}

\section{diss.tex}
{\scriptsize\verbatiminput{diss.tex}}

\section{proposal.tex}
{\scriptsize\verbatiminput{proposal.tex}}

\chapter{Makefile}

\section{makefile}\label{makefile}
{\scriptsize\verbatiminput{makefile.txt}}

\section{refs.bib}
{\scriptsize\verbatiminput{refs.bib}}


\chapter{Project Proposal}

\input{proposal}

\end{document}
```

A.2 proposal.tex

```
% Note: this file can be compiled on its own, but is also included by
% diss.tex (using the docmute.sty package to ignore the preamble)
\documentclass[12pt,a4paper,twoside]{article}
\usepackage[pdfborder={0 0 0}]{hyperref}
\usepackage[margin=25mm]{geometry}
\usepackage{graphicx}
\usepackage{parskip}


\usepackage[UKenglish]{babel}% Recommended
\usepackage[bibstyle=numeric,citestyle=numeric,backend=biber,natbib=true]{biblatex}
```

```
%\addbibresource{proposal_bibliography.bib}


\begin{document}

\begin{center}
\Large
Computer Science Tripos -- Part II -- Project Proposal\\[4mm]
\LARGE
Conflict Free Document Editing with Different Technologies\\[4mm]

\large
J.~Send, Trinity Hall

Originator: J.~Send

10 October 2016
\end{center}

\vspace{5mm}

\textbf{Project Supervisor:} S.~Kollmann

\textbf{Director of Studies:} Prof.~S.~Moore

\textbf{Project Overseers:} Prof.~T.~Griffin \& Prof.~P.~Lio

% Main document

\section*{Introduction}

\subsection*{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~\cite{Technion}. There are two main technologies that are used to enable concurrent editing of a
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~\cite{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 guaran
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 t
listed in later sections.

\subsection*{Resources required}

The primary external resource I will need is the Javascript library ShareJS, which is published under the MIT license on Git

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.
```

```
\subsection*{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.

\section*{Work to be done}

\subsection*{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 conclusi
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.

\subsection*{Detailed Project Structure}

\begin{enumerate}

\item \textbf{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.


\item \textbf{Implement Simulation:} Model having an arbitrary number of clients each running the CRDT algorithms, and simul
networking between these clients. Because this is P2P, it may be worth adding functionality for a variety of network
topologies.

\item \textbf{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.

\item \textbf{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.

\item \textbf{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.

\end{enumerate}



\subsection*{Possible extensions}

There are two extensions of varying difficulty:

\begin{itemize}

\item (Easier) Replace networking simulation with a P2P networking library. The end result of this extension should be a rea

\item (Difficult) Research prior work on undo and move functionality using CRDTs. If something suitable is found, implement

\end{itemize}


\section*{Success criteria}
```

```
These are the main success criteria associated with my project

\begin{enumerate}
\item A concurrent text editor based on CRDTs has been implemented.

\item The concurrent text editor passes all correctness tests.

\item Quantitative results comparing ShareJS and the CRDT based system have been obtained and analyzed.
\end{enumerate}


\section*{Timetable}

Planned starting date is 16/10/2011.

\begin{enumerate}

\item \textbf{Michaelmas weeks 2--3} Develop CRDT datastructure and algorithms on paper. Read into P2P networks
and simulating them.

\item \textbf{Michaelmas weeks 4--5} Lay out project files and implement network simulation with support for different
P2P topologies.

\item \textbf{Michaelmas weeks 6--8} Implement CRDT datastructures and algorithms, and connect these to network simulation.

\item \textbf{Michaelmas vacation} Learn an appropriate testing framework, write and generate unit tests for correctness of

\item \textbf{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.

\item \textbf{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.

\item \textbf{Lent weeks 5--6} Continue writing dissertation and optimizing system. Begin research for extension
which implements proper networking stack.

\item \textbf{Lent weeks 7--8} Continue writing dissertation. Before terms ends, review and peer-review (including superviso
incomplete draft. Implement networking extension. Research undo and move operations with CRDTs.

\item \textbf{Easter vacation:} Finish dissertation draft. Work on undo and move extensions for system.

\item \textbf{Easter term 0--2:} Edit and proof read dissertation. Work on extensions.

\item \textbf{Easter term 3:} Proof read and then submit early to concentrate on exams.

\end{enumerate}

\newpage

%\printbibliography

\end{document}


\end{filecontents}
```

Appendix B

Makefile

B.1 makefile

B.2 refs.bib

```
@misc{MotherDemo,
  title = {The Mother of All Demos, Reel 3},
  howpublished = {\url{https://archive.org/details/XD300-25_68HighlightsAResearchCntAugHumanIntellect&start=286}},
  note = {Accessed: 2017-03-01}
}

@article{Ellis1989,
author = {Ellis, C A and Gibbs, S J},
title = {{Concurrency Control in Groupware Systems}},
year = {1989}
}

@article{Gilbert2005,
author = {Gilbert, Seth and Lynch, Nancy},
pages = {51--59},
title = {{Brewer ' s Conjecture and the Feasibility of Consistent , Available , Partition-Tolerant Web Services}},
year = {2005}
}

@inproceedings{zeller2014,
  title={Formal specification and verification of CRDTs},
  author={Zeller, Peter and Bieniusa, Annette and Poetzsch-Heffter, Arnd},
  booktitle={International Conference on Formal Techniques for Distributed Objects, Components, and Systems},
  pages={33--48},
  year={2014},
  organization={Springer}
}

@inproceedings{preguica2009,
  title={A commutative replicated data type for cooperative editing},
  author={Preguica, Nuno and Marques, Joan Manuel and Shapiro, Marc and Letia, Mihai},
  booktitle={Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on},
  pages={395--403},
  year={2009},
  organization={IEEE}
}

@techreport{weiss2008,
```

```
  TITLE = {{Logoot: a P2P collaborative editing system}},
  AUTHOR = {Weiss, St{\'e}phane and Urso, Pascal and Molli, Pascal},
  URL = {https://hal.inria.fr/inria-00336191},
  TYPE = {Research Report},
  NUMBER = {RR-6713},
  PAGES = {13},
  INSTITUTION = {{INRIA}},
  YEAR = {2008},
  PDF = {https://hal.inria.fr/inria-00336191/file/main.pdf},
  HAL_ID = {inria-00336191},
  HAL_VERSION = {v3}
}

@article{weiss2010undo,
  title={Logoot-undo: Distributed collaborative editing system on p2p networks},
  author={Weiss, Stephane and Urso, Pascal and Molli, Pascal},
  journal={IEEE Transactions on Parallel and Distributed Systems},
  volume={21},
  number={8},
  pages={1162--1174},
  year={2010},
  publisher={IEEE}
}

@article{lamport1978,
  title={Time, clocks, and the ordering of events in a distributed system},
  author={Lamport, Leslie},
  journal={Communications of the ACM},
  volume={21},
  number={7},
  pages={558--565},
  year={1978},
  publisher={ACM}
}

@article{shapiro2007,
  author    = {Marc Shapiro and
               Nuno M. Pregui{\c{c}}a},
  title     = {Designing a commutative replicated data type},
  journal   = {CoRR},
  volume    = {abs/0710.1784},
  year      = {2007},
  url       = {http://arxiv.org/abs/0710.1784},
  timestamp = {Mon, 05 Dec 2011 18:05:29 +0100},
  biburl    = {http://dblp.uni-trier.de/rec/bib/journals/corr/abs-0710-1784}
}

@techreport{shapiro2011,
author = {Shapiro, Marc and Pregui, Nuno and Baquero, Carlos and Zawirski, Marek},
number = {RR-7506},
pages = {50},
type = {Research Report},
institution = {{INRIA}},
title = {{A comprehensive study of Convergent and Commutative Replicated Data Types}},
url = {http://hal.inria.fr/inria-00555588},
year = {2011},
HAL_ID = {inria-00555588}
}

@book{takada2013,
author = {Takada, Mikito},
pages = {1--62},
title = {{Distributed Systems: for fun and profit}},
year = {2013}
}

@misc{sharejs,
```

```
  author = {Gentle, Joseph},
  title = {ShareJS v0.6.3},
  year = {2013},
  publisher = {GitHub},
  howpublished = {\url{https://github.com/josephg/ShareJS/tree/0.6}},
  commit = {9291e9b1d2593565fdba6d45b96cd58b62d9b178}
}


@inproceedings{kumawat2016,
  title={Analysis of Operational Transformation Algorithms},
  author={Kumawat, Santosh and Khunteta, Ajay},
  booktitle={Proceedings of the International Conference on Recent Cognizance in Wireless Communication \& Image Processing}
  pages={9--20},
  year={2016},
  organization={Springer}
}

@techreport{RFC2460,
  author = {Stephen E. Deering and Robert M. Hinden},
  title = {Internet Protocol, Version 6 (IPv6) Specification},
  howpublished = {Internet Requests for Comments},
  type = {RFC},
  number = {2460},
  year = {1998},
  month = {December},
  issn = {2070-1721},
  publisher = {RFC Editor},
  institution = {RFC Editor},
  url = {http://www.rfc-editor.org/rfc/rfc2460.txt},
  note = {\url{http://www.rfc-editor.org/rfc/rfc2460.txt}}
}

@inproceedings{nedelec2013lseq,
  title={LSEQ: an adaptive structure for sequences in distributed collaborative editing},
  author={N{\'e}delec, Brice and Molli, Pascal and Mostefaoui, Achour and Desmontils, Emmanuel},
  booktitle={Proceedings of the 2013 ACM symposium on Document engineering},
  pages={37--46},
  year={2013},
  organization={ACM}
}

@inproceedings{nedelec2013,
  title={Concurrency Effects Over Variable-size Identifiers in Distributed Collaborative Editing},
  author={N{\'e}delec, Brice and Molli, Pascal and Mostefaoui, Achour and Desmontils, Emmanuel and others},
  booktitle={Proceedings of the International workshop on Document Changes: Modeling, Detection, Storage and Visualization,
  volume={1008},
  pages={0--7},
  year={2013}
}

@online{compsysmodeling,
author = {Dr~R.J.~Gibbens},
title = {Computer Systems Modeling Lecture Notes},
year = 2017,
url = {http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm}
}
```

Appendix C

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.