DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Real-time collaborative editing using CRDTs

JAKOB IVARSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Real-time collaborative editing using CRDTs

JAKOB IVARSSON

Master in Computer Science
Date: February 9, 2019
Supervisor: Johan Gustavsson
Examiner: Olov Engwall
School of Electrical Engineering and Computer Science

Abstract

Real-time collaborative editors such as Google Docs allow users to edit a shared document simultaneously and see each others changes in real-time. This thesis investigates how conflict-free replicated data types (CRDTs) can be used to implement a general purpose data store that supports real-time collaborative editing of semi-structured data. The purpose of the data store is that it can be used by application developers to easily add collaborative behaviour to any application.

The performance of the implemented data store is evaluated and the results show that using CRDTs comes with a performance and memory penalty. However, replication over the internet is very efficient and concurrent updates is handled in a predictable way in most cases.

Sammanfattning

Kollaborativa realtidseditorer som Google Docs låter användare editera ett gemensamt dokument samtidigt och se varandras ändrigar i realtid. Den här rapporten undersöker hur konfliktfria replikerade datastrukturer (CRDTs) kan användas för att implementera en generell databas som hanterar kollaborativ realtidseditering. Syftet med databasen är att den kan användas av applikationsutvecklare för att enkelt kunna lägga till kollaborativt beteende till applikationer.

Prestandan av den implementerade databasen utvärderas och resultaten visar att användningen av CRDTs resulterar i en en ökad minnesanvändning och sämre prestanda. Att replikera databasen är väldigt effektivt och den hanterar konflikter på ett förutsägbart sätt.

Contents

Chapter 1

Introduction

Collaboration through software is a core part of many peoples lives. We communicate by email and chat systems and work together with other people on documents, spreadsheets and presentations. The most advanced type of such collaborative software are those that support real-time collaborative editing, i.e. that multiple users can edit the same document simultaneously and see each others changes in real-time.

With the emergence of modern cloud computing and mobile applications, we typically need to support offline editing, i.e. the ability to continue using the application without having a network connection. These edits are later synchronized to the cloud and the other devices when the network comes back.

What these features have in common is that the application state needs to be replicated across multiple devices. Each replicated state can be updated locally and later synced to other devices. This replication requires a fundamental shift in the mindset of the application from only being a view of a server side state to being a node in a distributed system where each node has a separate state. This shift comes with a series of new challenges such as how to achieve consistency between nodes and how to handle conflicts that can arise when different nodes edit a shared state in parallel.

A popular example of a real-time collaborative product is Google Docs [14]. It achieves eventual consistency by using a technique called Operational transformation (OT) [15], which was introduced by Ellis and Gibbs in 1989 [12]. However, more recent advancements in distributed systems that promise strong eventual consistency are conflict-

free replicated data types (CRDTs), which was formulated by Shapiro et al. in 2011 [30]. CRDTs is a collective term for data structures that can be replicated across nodes and guarantees that any nodes that have received the same updates will end up in the same state.

1.1 Problem definition

This thesis project was carried out at Mentimeter [16] which is a company making a web based interactive presentation tool. This presentation tool is an audience response system that works by the presenter asking a question that the audience can vote on using e.g. their mobile phones. The presentation displays the voting results in real-time. Creating and editing presentations is also done in a web application. This project aims to implement a data store for real-time collaborative editing which can be used in their presentation editing software.

There are many layers to be considered when making a practical implementation of a collaborative data store (in order of relative significance for the project): the core algorithms and data structures that constitutes the implementation of the conflict-free data types; network, e.g. protocol and transport used; user interface, i.e. the APIs that will be exposed to the application developer; and persistence, how the data structures will be stored on disk (both on client and server). The core data structures will be the focus of this study and the main thing being evaluated.

While many collaborative systems (e.g. Google Docs) are free-text editors, Mentimeter presentations rely on a more structured data model. The different types of data structures imply an additional set of challenges regarding support of real-time collaborative editing. The data store implementation should specifically support representations of the most common data types such as List, Map, String, Number, Boolean and Null as well as nesting other data types in Lists and Maps. These are most of the primitive data types in Javascript and all the data types that can be encoded in JSON which is the most commonly used format for transferring data over the web. This type of data structures can be labeled as semi-structured data.

Basic operations on the data structures should also be supported, such as Add, Remove, Insert and Update. The data store should further be immutable, which is considered best practice for maintaining

application state and pioneered by libraries like Redux [26]. From the perspective of the application developer the data store should be opaque, meaning that the developer should not have to work with the underlying data structure and deal with the synchronization. Since Mentimeter is a web based product, a limitation is that any implementations is developed for the web and written in Javascript.

When evaluating the performance of the data store we care about the perceived latency by the user, the end-to-end latency as well as the scalability. User latency is the time it takes for a local update to take effect and end-to-end latency is the time from the update being initiated until it has taken effect on a remote replica. Scalability consists of how many concurrent users that can be supported as well as how the memory and latency grow in relation to the size of the data store.

1.2 Objective

The objective of this thesis is to research the computer science problems that need to be solved in order to build a real-time collaborative editing product that uses semi-structured data. The theoretical part of the project is to identify the requirements of such a product, identify challenges, explore different solutions and propose implementation strategies.

The practical part of the project is to design and implement a general purpose data store that supports collaborative editing and real-time synchronization between nodes over the network. The purpose of the data store is that it can later be used by application developers to develop a wide range of collaborative applications. For example, it could be used to keep the application state in sync across multiple devices.

1.3 Research questions

- How should existing conflict-free replicated data types be composed to support semi-structured data for real-time collaborative editing?
- What collaborative behaviour does the implemented data store support?

- How does the implemented data store perform compared to a non-CRDT solution in terms of runtime for operations, memory overhead of data structures and scalability?

Chapter 2

Background

Collaborative editing is an advanced feature of many document editing software. In such distributed systems, each node (for example a client application) has a replicated instance of the object being edited, called a replica. In order for the editing to be done in real-time, providing a responsive and interactive experience for the user, the local replica is first updated and the change is later sent to the other replicas. This approach is called optimistic replication [28] and comes with a series of challenges which are described in the following sections.

In order to build a real-time collaborative data store, these challenges need to be addressed. A general purpose data store should support a variety of data types, such as Map, List, Boolean, Number and String.

This chapter will discuss in detail the challenges with replicated data, the advances in the field and how it relates to this project.

2.1 Distributed systems theory

2.1.1 CAP theorem

The CAP theorem, introduced by Eric Brewer in 2000 [8] and later proven by [13], states that a distributed data store can guarantee two of the following properties:

- (C) Consistency. Every read in the data store always returns the latest written data or an error.
- (A) Availability. The data store always respond with a non-error.

- (P) Partition tolerance. The data store continues to function in the event of a network partition where messages are being dropped.

Thus, a distributed data store can be either a CP (consistent under partition) or AP (available under partition) system. CA (consistent and available) is not possible in a distributed system since we cannot guarantee the absence of network partitions. One example of a CP system is ACID databases. Examples of AP systems are the Domain Name System (DNS) and DynamoDB [11].

Although the CAP theorem was initially concerned with distributed data stores, it can also be applied to arbitrary replicated data systems, such as collaborative editing systems [7]. Such optimistic replication systems are by definition AP systems since we first write the data locally and then replicate the data. This means that another replica will return older data until it receives the new update.

2.1.2 Eventual consistency

The CAP theorem is a binary classification of consistency; a system is either consistent or not consistent. Vogel [32] introduces a more nuanced notion of consistency while discussing Amazon's infrastructure, the implementation of DynamoDB and the challenges and trade-offs in distributed systems. He classifies consistency in two categories, strong consistency and weak consistency.

Strong consistency is when every write is immediately reflected by all replicas. This is the property that CP systems have, but it comes with a penalty on performance and scalability.

AP systems on the other hand, by definition, have *weak consistency*. This means that if a write happens to a distributed data store, it is not guaranteed that the data written will be reflected in any subsequent requests.

There are stronger forms of weak consistency such as *eventual consistency*. For eventual consistency, the system guarantees that if a write has happened, the other replicas will eventually return the latest data when read.

Strong eventual consistency (SEC) [30] is an even stronger guarantee that any two replicated nodes that have received the same updates will be in the same state. This means that updates will automatically be merged and conflicts resolved without any external communication with other nodes and rollbacks.

2.1.3 Time, clocks, and the ordering of events

Knowing the order in which events happened is an important concept for resolving conflicts and providing eventual consistency. The exact order in which events happened in a distributed system is impossible to define[19]. However, it is possible to define a partial order of events. This partial order can be described by causality. An event A is said to have a causal relation to event B if B is dependent on A, i.e. B happened after A is observed. The events are said to be causally independent otherwise. In a distributed system, wall clocks, which represent physical time, cannot be used to keep track of causality. This is because there is no guarantee that the clocks in different nodes are accurate or run at the same speed. Instead, a number of logical clocks can be used to determine causality.

Lamport introduced a logical clock known as *Lamport clock* in 1978 [19]. To describe the clock he defines a process as a sequence of events. A system consists of a set of processes connected over a network. Processes can communicate by sending messages. Events can be categorized as internal events and sending and receiving messages. Each process in the system has a clock which associates each event with the current timestamp of the process. The clock is a counter which increments each time an event occurs in the process. When a process sends a message it contains the current timestamp of the process. When another process receives this message, if its clock is less than the message timestamp, it sets its own clock to be greater than the timestamp. Lamport clocks thus provide a partial order of events. We know if some events happened before others. However, two events from different processes can still have the same timestamps. To define a total order, one can use tie breaking rules such as giving each process a unique sortable id and order the events by process id if the timestamps are equal.

Lamport clocks have some undesirable properties. One problem is that the algorithm does not preserve the causal independence between events since it assigns a linear order to all events. For example, two concurrent events from different processes have different timestamps and thus it appears like one event came before the other even though there is no causal relation between the events. There are multiple algorithms that can capture this causal independence. Two of the most commonly used are version vectors [23] and vector clocks [21], which

only differ in small details. The vector clock can be seen as a vector of Lamport clocks, one for each process. A timestamp is an instance of this vector. When an event occurs in a process, it increments its own component in the vector. When a process receives a message, it takes the timestamp of the message and takes the max of each component with its internal vector clock. This gives us is an improved ability to determine causality. An event is causally dependent on another event if all components in its vector clock is less than or equal to the components in the other vector. Conversely, if neither event is dependent of the other then the events are causally independent which means that the events happened concurrently. This property makes vector clocks more suitable in situations where such information is needed. However, it comes at the cost of the having to store a vector of each process instead of a single counter. The entire vector also needs to be present in each message sent which can add a significant overhead as the number of processes grows. There are solutions to these scalability issues such as Concise Version Vectors [20], Interval Tree Clocks [2] and Dotted Version Vectors [25]. However, all solutions come with their respective tradeoffs.

2.2 Collaborative editing approaches

This section describes two of the most well-known approaches to implement collaborative editing systems, operational transformation and conflict-free replicated data types.

2.2.1 Operational transformation

Operational transformation (OT) is one of the first collaborative editing approaches and was described by Ellis and Gibbs in 1989 [12]. It works by a central server which keeps track of the state that each client is in. When a client updates the state, the node optimistically executes the operation locally. The operation is then sent to the server which transforms the operation for each other client so that all clients end up in the same state after executing the operation. The transformed operations are sent to the clients which execute them in their local state. Conflicting updates are resolved using tie breaking rules.

A number of OT algorithms for collaborative text editing have been

developed, for example GOTO [31]. One of the most well known applications of OT is Google Docs [15].

All OT algorithms require a central server that transform the operations for each client. The algorithms are also complex to implement and use *ad hoc* rules to resolve conflicts. A peer-to-peer solution for collaborative text editing called WOOT (WithOut Operational Transform) was presented in 2005 [22]. The solution was simpler than previous OT algorithms and is formally proven to be correct. WOOT created a new direction for collaborative editing techniques which were later labeled as conflict-free replicated data types.

2.2.2 Conflict-free replicated data types

Conflict-free replicated data types (CRDTs) were formally introduced in 2011 [30]. CRDTs are replicated objects that provide SEC. Assuming eventual delivery, CRDTs guarantee that they converge to the same state. Unlike OT algorithms, CRDTs do not require a central server that transforms the operations and can thus be used for peer-to-peer replication of data. CRDTs need to fulfill a set of mathematical properties and can therefore be proven to be correct. CRDTs have existed long before the concept was formalized and use cases have been found in distributed file systems and databases, among others.

CRDTs can be categorized into two general types, state-based and operation-based. The main difference is the way that the data type replicates its state. The state-based version applies operations locally and then send the new state to the other nodes for replication. The operation-based version instead sends the operations themselves for replication.

Shapiro et al. have shown that state-based and operation-based CRDTs are equivalent and can be emulated by each other [30].

State-based CRDTs

State-based CRDTs, also known as convergent replicated data types, send the full state to replicate data [29] [30]. On receiving a remote state, the node merges the remote with the local state. In order for the replicas to converge to the same state, the merge function must fulfill the following properties:

- Commutative: $f(a, b) = f(b, a)$

- Associative: $f(a, f(b, c)) = f(f(a, b), c)$
- Idempotent: $f(f(a)) = f(a)$

where f is the merge function a and b are two states.

The key characteristic of state-based CRDTs is that the number of transmissions and delivery order does not matter, the data will still be SEC. However, one issue with state-based CRDTs is that they can be inefficient if the data transmitted is big. There are solutions to this such as delta state replication [3], but they require complex protocols to resolve what needs to be transmitted in order to converge.

Operation-based CRDTs

Operation-based CRDTs, also known as commutative replicated data types, replicate by sending individual operations over the network [29] [30]. An operation-based CRDT has to fulfill the following properties:

- Operations have to be delivered exactly once in causal order to all replicas.
- Causally independent operations have to be commutative when applied to the local state.

The main benefit of operation-based CRDTs is the small amount of data is sent over the network compared to state-based. However, the implementation of the delivery mechanism is a lot harder. The efficient use of the network makes operation-based CRDTs preferable for real-time collaborative editing since we want to send small incremental updates frequently in order to achieve a real-time feeling of the application.

2.2.3 Other approaches

Another approach to achieve eventual consistency is Cloud Types [10]. Cloud Types have a lot in common with CRDTs but are specifically made for replicating cloud data with a mobile client and is not made for collaborative editing.

The Bloom language have gone for a completely different approach and has implemented a complete programming language with a powerful consistency analyzer using the CALM theorem [4]. This approach

could not be used in this project since the programming should be done in Javascript.

2.3 Examples of CRDTs

This section describes different types of CRDTs found in the literature.

2.3.1 Register

One of the most basic types of CRDTs is registers. A register is a data type that holds an immutable value. The value could be of any type, for example Boolean or String. The only operation a register supports is *update* which replace the current value with a new one. Registers are not conflict-free by default since updates that happen concurrently lead to conflicts when trying to reconcile. The fundamental problem is how to choose which update to keep and which to discard. To resolve this we need to know the order of updates and which updates happened concurrently. How to do this in a distributed system is described in subsection 2.1.3.

Last-Writer-Wins Register

Using a Lamport or vector clock and a unique id for each replica, a Last-Writer-Wins Register (LWW-Register) [29] is trivial to implement. The LWW-Register stores the value together with a timestamp. A state-based register is replicated by sending the entire state. When receiving an update by a remote replica the two registers are merged by choosing the one with the greatest timestamp.

Multi-Value Register

An alternative to the LWW-Register is the Multi-Value Register (MV-Register) [29]. It works almost the same as the LWW-Register but in the case of a concurrent update, instead of choosing one of the values both are stored in the register. It is then up to the application developer to merge these two versions. On the next successive update, the multiple values in the register are replaced with a single new value. A famous use case of a MV-Register is in Amazon's Dynamo database [11]. Note that implementing a MV-Register using a Lamport clock

does not make much sense due to its weak notion of causal independence, using a vector clock would be more effective.

2.3.2 Set

A Set is a basic data type that can be used as a building block to implement other, more complex, data types such as Maps, Lists and Graphs [29]. A Set usually supports *add* and *remove* elements. Such a set is not conflict-free by default since the remove operation is not commutative.

Maps will be the primary use-case for this project. However, maps can be seen an instance of set where each element is a key-value pair.

Grow-Only Set

The simplest form of CRDT Set is the Grow-Only Set (G-Set) that only supports the add operation. In the state based version, the merge function simply takes the union of the two sets. This is a CRDT since the union function is commutative, associative and idempotent.

Two-Phase Set

A Two-Phase Set (2P-Set) is a CRDT Set that supports both adding and removing elements. It combines two Grow-Only Sets, one for added elements and one for removed. The removed set is also called a tombstone set and is required in ordered for the 2P-Set to converge. A caveat with the 2P-Set is that once an element has been removed it can never be added again.

Last-Writer-Wins Set

The Last-Writer-Wins Set (LWW-Set) takes inspiration from the LWW-Register and 2P-Set. It associates each added and removed element with a timestamp. An element is in the set if it does not have a corresponding element in the tombstone set with a later timestamp. By using a vector clock, concurrent add and remove operations can be detected and preferred semantics for merging can be chosen such as remove-wins or add-wins.

Observed-Remove Set

The Observed-Remove Set (OR-Set) [29] closely resembles the 2P-Set but with the ability to re-add a previously removed element. This can be done by generating a unique id when adding an element and storing the element and id together. This makes each add unique and thus when removing an element, only the element with the corresponding id will be placed in the tombstone set. This creates an add-wins semantic for concurrent add and remove operations, i.e. since each removed element is associated with an id adding the same element with another id will take precedence.

An optimized version of the OR-Set [6] without using tombstones can be achieved by using a communications middleware that preserves causality. By making sure that the add comes before the corresponding removal of the same element, given that each add is unique we can simply remove the element from the set completely.

2.3.3 List

A List is an ordered sequence of elements. The list structure is commonly used for collaborative text editing and supports adding elements in relation to other elements with *addRight* and *remove*.

Replicated Growable Array

The Replicated Growable Array (RGA) [27] is a total ordered sequence. It stores added and removed elements together with timestamps in a 2P-Set and an adjacency list of edges between those elements in a G-Set. The edges form a tree which when traversed in pre-order gives the totally ordered sequence. If two elements are added to the right of the same element in the sequence then they are ordered by timestamp, with the highest timestamp to the left of the other. Concurrent additions can be ordered by some tie-breaking rule like process id.

Tree-based sequences

Other tree based sequences are Logoot [33] and TreeDoc [24]. Both CRDTs are specifically made for text document editing since they include for example lines in the implementation. This makes it harder to use for a general purpose list container. A comparison between RGA,

Logoot and TreeDoc and a few other sequence CRDTs shows that RGA and Logoot on average perform best for text editing [1].

2.3.4 Partially-Ordered Log

The Partially-Ordered Log (PO-Log) [5] is a CRDT that defines a partial order over operations. Operations are stored instead of values in the log together with corresponding timestamps in a 2P-Set. Using a vector clock to generate timestamps we can get a partial order of the operations in the set. The PO-Log CRDT can be used to implement other CRDTs such as OR-Set and MV-Register by defining algorithms that compute a view of the log [5]. The PO-Log is purely operational since only operations are stored. However, it can quickly become inefficient to store all operation made. To mitigate this, a log compaction algorithm can be implemented to remove operations that are no longer necessary to compute the final state.

2.3.5 Embedding CRDTs

Allowing application developers to embed (or nest) data structures in each other is important when desigining a general purpose data store since it allows for composition and more flexible data modelling. One example of CRDTs that supports embedding is the Map type in Riak [9]. The Riak Map can embed registers, counters, flags and other maps and allows for updating the embedded CRDTs.

Another CRDT that allows for embedding is the Conflict-Free Replicated JSON Datatype, introduced by Kleppman and Beresford in 2017 [18]. The JSON CRDT is operation-based and supports all JSON data types (Number, Boolean, String, List and Map) as well as embedding other data types in Lists and Maps. It uses MV-Registers to keep conflicting changes and lets the application developers resolve conflicts. The specification uses a list of all preceding operations that an operation depends on to keep track of causality and identify concurrent operations. This list can in theory become infinitely big as it grows on each operation. However, the authors suggest that a vector clock or an optimized version of it could be used instead. The JSON CRDT also handles merging of concurrent edits in a manner which can lead to what application developers might consider invalid states. The basis for composition is complex and requires advanced manipulation of

indices and cursors. Data is never deleted from the state but it instead keeps a presence Set of data that is active. These things should be fixed in a practical implementation.

2.4 Requirements

Given the problem definition described in section 1.1 and the different solutions described in this chapter we have the following requirements for the collaborative data store (in priority order):

- It should be immutable and written in Javascript.
- It should have strong eventual consistency.
- It should be able to store the basic data structures List, Map, String, Number, Boolean and Null as well as nesting other data types in Lists and Maps.
- Conflicts should be automatically resolved and not handled by application developers.
- It should be scalable. There should be no hard limit for the number of nodes than can use the data store. It should be efficient to use over time and to store large amounts of data.
- It should be efficient to replicate over the network.

2.5 Summary

CRDTs is the current state of the art to handle data replication in eventually consistent systems and collaborative editing.

An operation based approach seems to best fit our use case since it allows for efficient synchronization over the network. However, it imposes stronger requirements on the message delivery.

In general CRDTs comes with a penalty in the storage size of the data structure. Most notably, removed elements are kept in the state as tombstones for some CRDTs in order to support concurrent updates. This is not ideal if we want to implement a scalable data store. However, we can avoid tombstones in some cases by making sure

that all added elements are unique if we have causally ordered delivery. Causal delivery can for example be implemented by using vector clocks or by using a central server to route the messages with a reliable communication protocol.

Some CRDTs also need to keep track of vector clocks, which grows with each new process. This can make CRDTs more inefficient over time, as each new process that edits the data structure will grow it indefinitely. Using vector clocks is not ideal for this project since we do not want to limit the number of nodes that can use the data store.

With the basic CRDT types of registers, sets and ordered lists we can store all of the required data structures and they should be enough for implementing the data store. We want to allow nesting inside lists and maps since the store should be general purpose. There is very little literature that describes composing CRDTs. The JSON type shows that CRDTs can be composed and used together in a manner similar to that we want. However, it has some limitations that we would like to avoid in our implementation such as the use of vector clocks and tombstones to make it more scalable and the odd collaborative behaviour. Also, since we do not want conflict resolution to be handled by application developers, using a LWW-Register is preferred over a MV-Register. For these reasons the decision was made to develop a new implementation that composes the basic CRDTs list, set and register.

In summary, using CRDTs we can implement a general purpose data store that support optimistic replication with strong eventual consistency guaranties which can be used for real-time collaborative editing.

Chapter 3

Method

The project consisted of implementing the data store which is described in section 3.1. The collaborative behaviour was evaluated in a few representative test scenarios. Finally, the performance was evaluated by running benchmarks described in section 3.3.

3.1 Specification

The implemented data store is general purpose and can store a wide range of data models using all of the data types that JSON supports. The implementation consists of two container types, list and map which both can contain other containers as well as primitive types. The primitive types such as String, Number, Boolean and Null are stored in LWW-Registers together with a Lamport timestamp. Every created object is associated with a unique identifier and lists and maps only store identifiers to other objects. The store handles composing all objects together and has all the APIs that the application programmer interacts with. To guarantee strong eventual consistency, all operations are applied in causal order. The exact implementations can be found as an open source library on Github [17]. The implementation is written in Javascript but specification is language independent.

3.1.1 Map

The map implementation is described in Specification 1. No specification of a map CRDT can be found in the literature. However, it can easily be implemented using a CRDT set. The map implementation

Specification 1: Map

```
1 state: {}
2 get (key):
3     return state[key]
4 add (key, id, timestamp):
5     if ¬∃state[key] or timestamp > state[key].timestamp:
6         state[key] = (id, timestamp)
7 remove (key, id):
8     if state[key].id = id:
9         delete state[key]
```

builds on the ideas of OR-Set but every value is associated with a key. The value is a unique id which is added together with a timestamp. The timestamp is a Lamport timestamp followed by a unique user id. The id is always unique and values are removed by id and key. This means that we do not have to store tombstones for removed elements and elments can be removed permentently. If an element is added to a key which is already associated with a value, we compare the timestamp with the new timestamp we want to add and if it is smaller we replace the old value with the new value. This works essentially like a LWW-Register. The main thing that we have solved here is concurrent assignments to the same key, which makes this map a CRDT.

3.1.2 List

The list implementation described in Specification 2 is based on the RGA algorithm with some modifications. It consists of an adjacency list of edges *E* and an OR-Set of vertices *V*. The adjacency list is a grow-only set of (*fromId*, *toId*, *timestamp*) tuples, where *fromId* is the id of the start node, *toId* is the id of the end node and *timestamp* is a Lamport timestamp of when the edge was created. This timestamp is used as an unique id for the edge. New elements are inserted by adding a new tuple to *E*. The edges create a tree structure and *values* evaluates this structure to create an ordered sequence of elements.

We introduced a new operation for the list, move, which changes the position of a list element. This operation is interesting since it allows for concurrent updates of the element being moved. It can be used for example in a todo list application to reorder the items. Move

Specification 2: List

state: set E, set V
values ():
    $sortedEdges$:= E first sorted in topological order starting from __$startId$__ then by timestamp
    $values$:= $\{e \in sortedEdges \mid \forall f \in E(e.toId = f.toId \Rightarrow e.timestamp > f.timestamp), \exists id \in V(id = e.toId)\}$
    return $values$
get (index):
    return values()[index]
insert (afterId, elementId, timestamp):
    $E := E \cup (afterId, elementId, timestamp)$
    $V := V \cup (elementId)$
move (afterId, elementId, timestamp):
    $E := E \cup (afterId, elementId, timestamp)$
remove (elementId):
    $V := \{v \in V \mid \neg(v = elementId)\}$

is implemented by creating a new edge for the element id that one wants to move and only the edge with the largest timestamp for each element is valid. Removing an element is done by removing the corresponding vertex from V. No tombstones are needed since all values in V are unique. However, edges are never removed from E since we need to keep the tree of edges intact, i.e. no orphaned nodes in the tree. There is a hard coded id for the first element in the list, __$startId$__, which is used to insert elements at the start of the list (position 0).

3.1.3 Store

The store is described in Specification 3. The store consists of a table s of CRDT objects, keyed by id. The store is initialized with a value, normally a list or map container, hard coded to the id __$rootId$__.

prepare transforms the user operation to a CRDT op that can be evaluated and applied to the store. The user operation consists of a path to the object that is to be updated, the operation that should be performed on this object and an argument to that operation. The path is a list of map keys and list indices that are used to lookup the object to be updated, starting from __$rootId$__. All supported operations are:

Specification 3: Store

```
1  state: set S , uid pid, lamport clock c
2  init (root): S := {__rootId__ : root}
3  prepare (path, opType, value):
4      op := {}
5      op.type := opType
6      op.objectId := lookup id that corresponds to path starting
         from __rootId__
7      op.params := get specific params for opType and object type
8      op.value := value
9      increment(c)
10     op.timestamp = timestamp(c, pid)
11     return op
12 apply (op):
13     c := max(c, op.timestamp)
14     object := register[op.objectId]
15     S[op.objectId] := object.perform(op.type, op.params)
16     if op.type = ADD or op.type = INSERT
17         S[op.timestamp] := op.value
18 gc ():
19     traverse object tree starting from __rootId__, marking objects
         as reachable
20     iterate through all objects in S and remove objects that are
         not marked as reachable
```

map add, map remove, list insert, list move, list remove and register set (also known as update). The prepare function is only evaluated at the source replica and the generated operation is asynchronously sent to the other replicas over the network.

apply takes a generated operation and the current state and returns a new state. The apply operation is evaluated on every replica. This is known as the downstream operation and is named *effect* in some literature [29].

gc is a garbage collection function that removes objects that no longer have any reference to them. This happens when a list or map is removed that contains nested objects. The algorithm is based on a simple mark and sweep algorithm. The function should run periodically to decrease the memory size of the data store.

3.2 Qualitative evaluation

The qualitative evaluation of the data store was done in two parts. The collaborative behaviour was evaluated first and then complexity analysis was performed on the implementation.

3.2.1 Collaborative behaviour

To evaluate the collaborative behaviour of the data store, a few example scenarios were chosen that highlight the features of the data store. Inspiration for the scenarios and how they are presented was taken from the JSON CRDT paper [18]. Some of the scenarios in the JSON CRDT paper are handled the same way by the implemented data store and some are handled differently. The scenarios chosen for this thesis are unique for the implemented data store and different from literature. For example, some of the scenarios describe the move operation which is not present in any literature.

The scenarios consist of two replicas that starts in the same state, concurrently apply operations and then synchronize the state. The purpose is to show how concurrent operations and conflicts are handled by the data store. The number of possible scenarios are infinite and therefore the scenarios in this thesis are only a small subset. In total, six scenarios were chosen that show the collaborative behaviour of lists and maps as well as nesting data in these containers. All of the

supported operations are used in these scenarios with the exception of List remove which is analogous to Map remove.

3.2.2 Complexity analysis

The implemented algorithms were analyzed for time and space complexity. The list, map and register implementations were analyzed. The functions of the data structures were analyzed for worst case time complexity. The space complexity was analyzed by taking the memory used by the data structures in relation to the number of elements stored.

The Javascript code was analyzed and not the specification since a number of optimizations were made. In the analysis, all immutable update operations are considered constant. The purpose of the analysis is to show the theoretical scalability of the data store, i.e. how the runtime and memory grow with the input size. The actual performance can be significantly different which is why the quantitative performance was evaluated as well.

3.3 Quantitative evaluation

This section describes the quantitative evaluation of the data store. The implementation was evaluated using a number of metrics derived in subsection 3.3.1. These metrics were then measured during a series of performance benchmarks, simulating the different use cases of the data store.

3.3.1 Performance metrics

The main concerns regarding performance was two-fold: end-to-end latency and scalability, see section 1.1. The local update should also not have too large latency since we do not want the user to feel that the application is unresponsive. End-to-end latency is the time it takes from one process starting to update the store to another process receiving that update and displaying it.

$$\textit{End-to-end latency} = \textit{Network latency} + \textit{Computation latency} + \textit{Render latency}$$

Rendering latency in terms of the data store is the time it takes to get the contents of the data store. To display the content in a UI depends

on the rendering engine (browser) and is out of scope of this project since it is not related to data replication. Getting the contents of the data store depends on the type of data stored since it is different for different data types.

Network latency is the time the message takes to travel from node A to node B over the internet. Very simplified, it can be described as follows:

$$Network\ latency = Payload\ size * Bandwidth$$

We cannot control bandwidth since it is dependent on the user's network, so what we have left is payload size which we can measure.

Computational latency for the data store can be calculated as the sum of the latency of the prepare and apply function:

$$Computational\ latency = Prepare\ op + Apply\ op$$

Prepare op does only have to be performed at the source, therefore it scales much better compared to apply. However, it still affects perceived user latency and end-to-end latency. Prepare and apply is different for different CRDTs and can depend on the size of the objects to update. Therefore, they should be measured independently.

Scalability consists of the number of operations per time unit, which could translate to the number of concurrent users, and the size of the data store. The size of the data store is dependent on the meta data overhead times data size. The data is up to application developers so the meta data overhead is what we are interested in.

In summary, the metrics we have are:

- Payload size.
- Execution time for prepare and apply.
- Memory overhead or size of data store over time.

3.3.2 Performance benchmarks

The benchmarks used to measure the performance of the data store are designed to show how the data store scales with the number of operations performed. There is one benchmark for each operation that the data store supports. Every benchmark consists of 10 000 operations that are executed sequential by the data store. Execution time is measured for each operation by first measuring the time for the *prepare*

function to execute and then measuring the time for the *apply* function executed with the results of prepare. The functions were timed using the Node.js function *process.hrtime*.

In order to evaluate the metrics, a baseline was implemented using immutable.js, without any collaborative functionality. The baseline simply takes an operation and updates the local data structure using the immutable.js operations that correspond to the data type. The execution time for this update function was measured to compare with the *prepare* and *apply* exection times.

Memory as well as payload size were measured after each operation had been applied to the data store. Memory is measured by running *JSON.stringify* on the entire data store and taking the length of the resulting string. The same process is performed to measure payload size by taking the operation returned from *prepare*. The memory size is measured similarly for the baseline object. Internally, the Immutable.js data structures are converted to plain Javascript objects before being serialized to JSON.

Following are descriptions on each benchmark:

List Insert starts with an empty list and inserts 10 000 elements at random positions in the list.

List Move starts with a list of 1000 elements and then randomly moves elements 10 000 times.

List Remove starts with a list of 10 000 elements and removes at random elements in the list until it is empty.

Map Add starts with an empty map and adds 10 000 values to different keys.

Map Remove starts with a map of 10 000 key-value pairs and removes them one by one.

Register Set starts with a map of 10 000 key-value pairs and performs 10 000 updates on random keys.

3.3.3 Technical details

The code is written in Javascript, using Immutable.js version 3.8.2. The benchmarks were executed in Node.js version v8.11.1. Computer

specs: Macbook Pro Early 2015, macOS Sierra 10.12.2, Intel Core i5-5257U (2.70 GHz), 8GB RAM.

Chapter 4

Results

The practical objective of this thesis was to design and implement a real-time collaborative data store using CRDTs. This chapter describes how this data store was evaluated. The qualitative part of the evaluation consists of examining the collaborative behaviour and a complexity analysis. The quantitative evaluation is a set of benchmarks which describes the performance characteristics of the data store.

4.1 Collaborative behaviour

The collaborative behaviour was evaluated qualitatively using a set of example scenarios. The purpose of this evaluation is to highlight the collaborative features of the real-time collaborative data store, i.e. to show how collaboration on shared data between different participants works using the data store.

Figures 4.1 to 4.6 show how the data store handles different but representative scenarios where replicas concurrently update the shared state. Every figure represents two replicas, *Replica A* and *Replica B*, starting in the same exact state. The replicas then perform concurrent updates and transition to a new state. Every state is represented as a box with the serialized data printed inside. Between each state is either a user performed operation or network communication. Network communication is represented by the dashed lines where the previous updates are transmitted and applied to the other replicas.

The first three scenarios describe operations on maps. Figure 4.1 shows how adding values to the same key is handled. After replication, only one of the values for the *theme* key is kept, in this case *Replica*

B's given that its id is greater than *Replica A*'s. This is due to the design of the data store where concurrently added maps are given different ids and can therefore not be merged. The second update is referencing this id and can therefore not be reconciled if the id is not present.

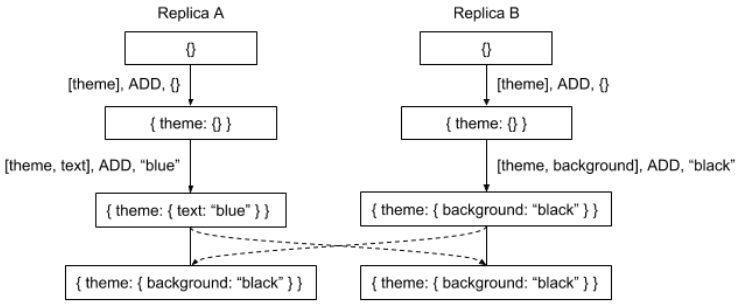


Figure 4.1: Concurrently add a map to the same key. The figure displays two replicas starting in the same state. Each box is a new state and the arrows are transitions to new states through applying the operations described next to the arrows. The dotted line is network communication.

Figure 4.2 shows a similar scenario to Figure 4.1 but two different keys are added to the same map. In this case, both of the updates are merged together and both keys are kept in the final map. In both scenarios conflicts are handled automatically but the second is ideal since no updates are lost in the final state.

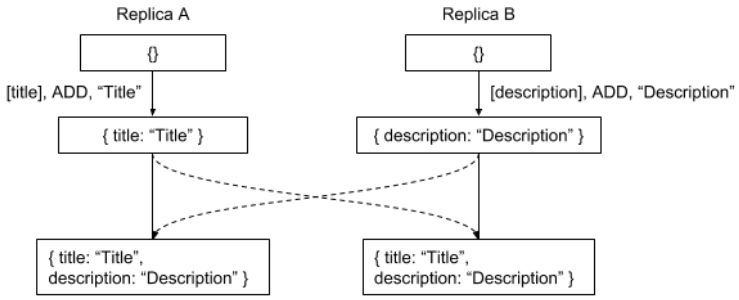


Figure 4.2: Concurrently add values to different keys in a map.

The last map scenario, see Figure 4.3, shows how concurrently removing and performing a nested update on the same key is handled. In this case the nested update is lost and the key is removed, which is expected.

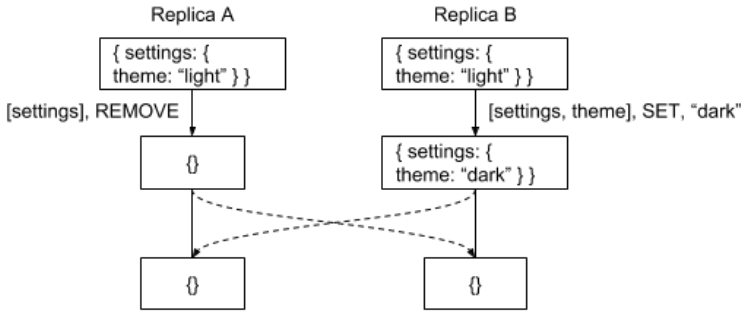


Figure 4.3: Concurrently remove and update the same key in a map.

The last three scenarios describe operations on lists. Figure 4.4 shows how concurrently moving and updating the same list element is handled. After replication, both of the updates, the new value and the new position of the list element, are present in the final list. This is the purpose of the move operation, that nested updates are kept when changing the position of an element. The move operation is unique to our implementation and not found in any literature.

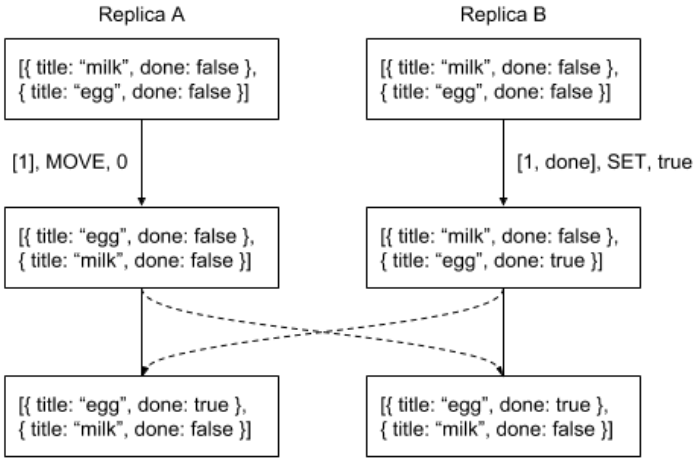


Figure 4.4: Concurrently update and move the same list element.

The next scenario displayed in Figure 4.5 describes how collaborative text editing is handled using a list where each letter is a list element. What makes this data structure usable for text editing is that it keeps the same inserted-after-element relationship to the original update after replication. Therefore, if multiple elements are inserted after

each other in the list then they will end up in that order after replication even though there are concurrent insertions at the same positions. In this case, the elements inserted by *Replica B* end up before the ones inserted by *Replica A* given that *Replica B*'s id is greater.

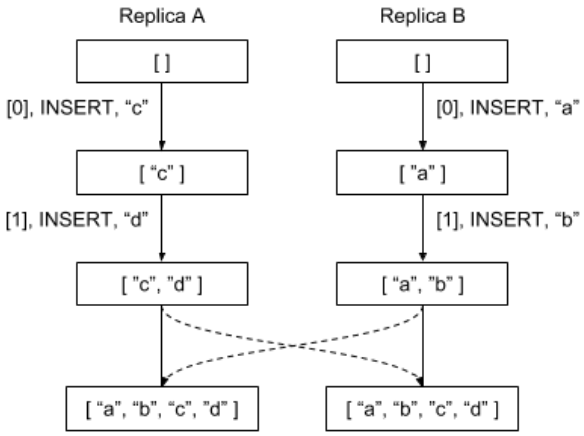


Figure 4.5: Concurrently insert elements in a list.

The last scenario in Figure 4.6 shows concurrently inserting and moving different elements in a list. The interesting property here is that even though "c" is inserted after "b", when "b" is moved "c" does not follow but ends up after "a" instead. This is due to the modifications to the original RGA algorithm to support the move operation, described in Specification 2. Instead of inserting after an element id we insert after an edge id. A new edge is created for the moved element but the inserted element is still inserted after the old edge. The three list scenarios together show that the list can be used for many different applications.

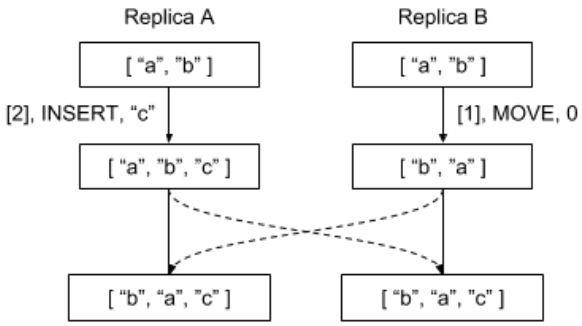


Figure 4.6: Concurrently insert and move different elements in a list.

4.2 Complexity analysis

We denote by:

V the vertices in the list. Insert creates a new vertex.

E the total number of edges created in the list. Both insert and move operations creates an edge. The edges are not removed when removing an element in a list.

A the added elements, e.g. key-value pairs in a map and elements in a list. Removed elements are not included in A.

P the payload size in a register.

Table 4.1 describes the worst-case time-complexity for the different operations. It is assumed that all the immutable updates are constant, $O(1)$. The List has overall worse time-complexity compared to Map and Register which both have constant time for all methods. Except for List Remove, all List operations grow linearly with the number of elements. This means that the List scales worse compared to the Map and Register.

Operation	Time-complexity
List Get	$O(E)$
List Insert	$O(E)$
List Move	$O(E)$
List Remove	$O(1)$
Map Get	$O(1)$
Map Add	$O(1)$
Map Remove	$O(1)$
Register Get	$O(1)$
Register Update	$O(1)$

Table 4.1: Worst-case time-complexity analysis.

Table 4.2 describes the space complexity for the different data types implemented, i.e. the total space taken with respect to the number of elements in the data structure. The only metadata overhead for the data types is the edges stored in the list.

Data type	Space complexity
List	$O(V + E)$
Map	$O(A)$
Register	$O(P)$

Table 4.2: Space complexity analysis.

4.3 Performance benchmarks

The performance of the data store was evaluated quantitatively using a series of benchmarks. There is one benchmark for each operation type that the data store supports, described in figures 4.7 to 4.12. A detailed description of how the benchmarks are implemented and the metrics used can be found in section 3.3.

Each benchmark consists of 10 000 operations executed sequentially. Execution time was measured for each operation. *prepare* and *apply* are the respective functions of the data store and *baseline* is the corresponding immutable.js function. After each executed operation, the memory of the data store and the baseline were measured. Finally, the payload size of the operation generated by *prepare* was measured.

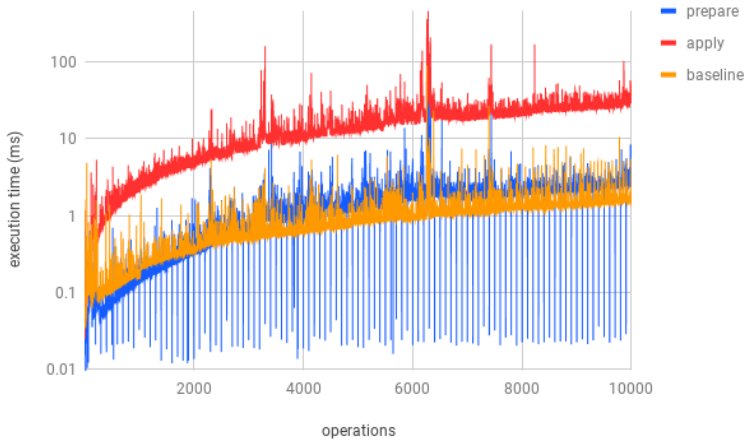
The execution time for insert, see Figure 4.7, grows linearly with the number of operations for all functions where *apply* has a a constant overhead compared to the other two. The large dips in execution time for the *prepare* function can be explained by inserting at position 0, since we do not have to lookup the id to insert after in this case. Even after 10 000 inserted elements the execution time remains acceptable at around 50 ms. More than that would noticeably impact the responsiveness of an application.

The move benchmark, see Figure 4.8, is interesting since the execution time for *apply* and *baseline* remains constant while *prepare* grows linearly. This growth is correlated with the memory of the data store which increases after each executed operation while the baseline memory remains constant.

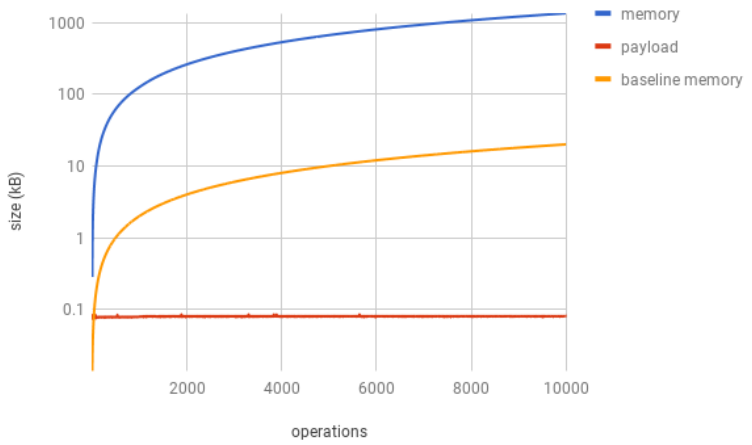
The list remove benchmark, see Figure 4.9, shows the downside of the list implementation. The execution time for *apply* and *baseline* decreases as elements are removed. However, *prepare* decreases at a much slower rate and this is once again correlated with the memory size. Even at the end of the benchmark, when there are no elements left in the list, the memory size is still very large. This is due to the edges between list elements are never removed.

The scalability for all map and register functions is very good, the execution time remains constant as the number of elements grow and there is nearly no overhead compared to the baseline implementation. The data store has a constant overhead in memory compared to the baseline but this is expected.

The payload size for all operations remains constant at around 100 bytes which means that they will be very efficient to send over the network.

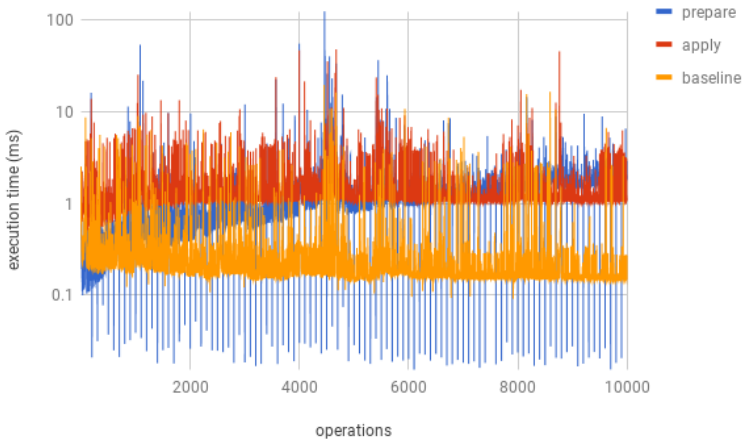


(a) Execution time

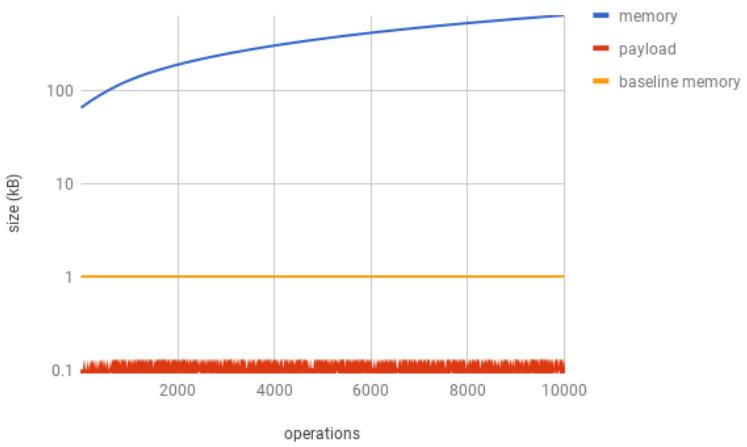


(b) Memory

Figure 4.7: Benchmark executing 10000 list insert operations sequentially. Execution time is measured for each operation along with *memory* size after the operation is applied and *payload* size of the operation generated by *prepare*. *prepare* and *apply* are the respective functions of the data store and *baseline* is the immutable.js function corresponding to the operation.

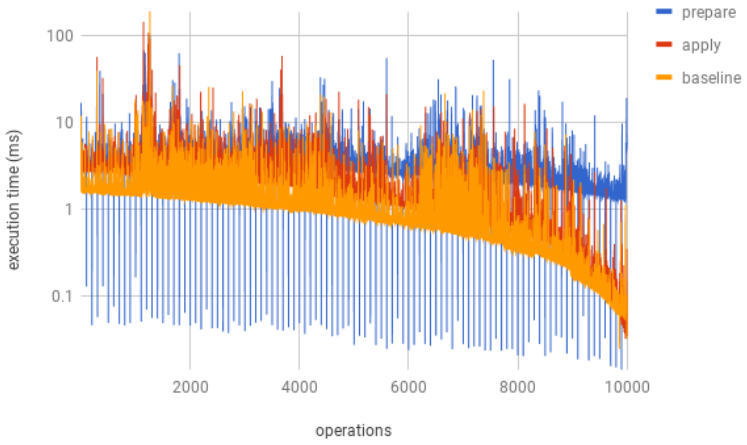


(a) Execution time

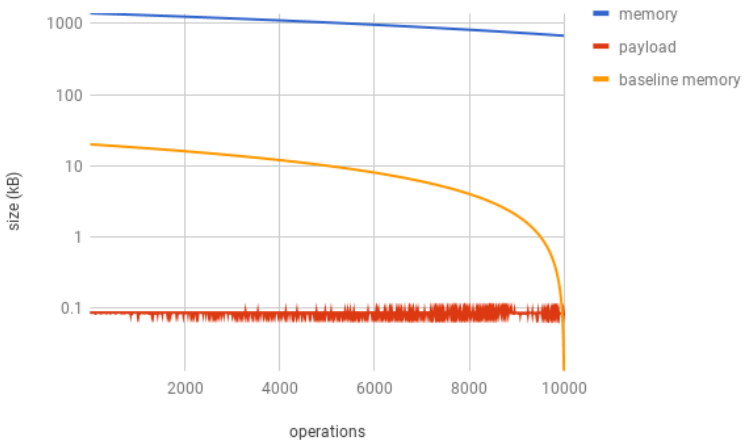


(b) Memory

Figure 4.8: Benchmark executing 10000 list move operations.

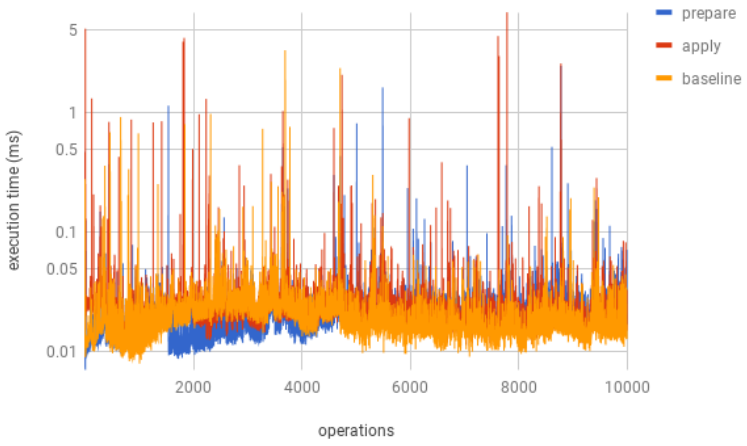


(a) Execution time

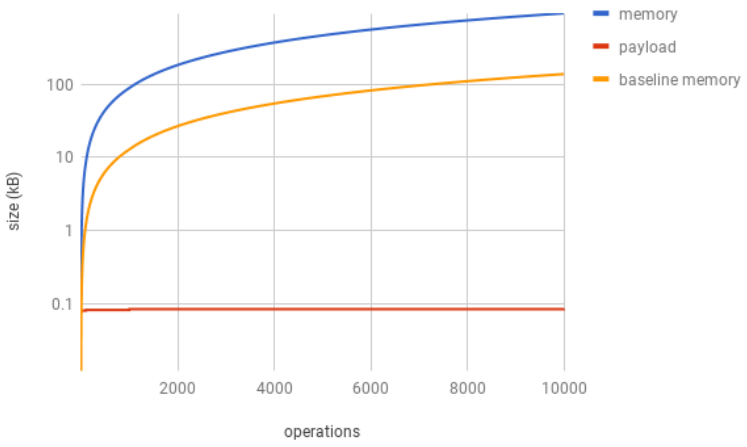


(b) Memory

Figure 4.9: Benchmark executing 10000 list remove operations.

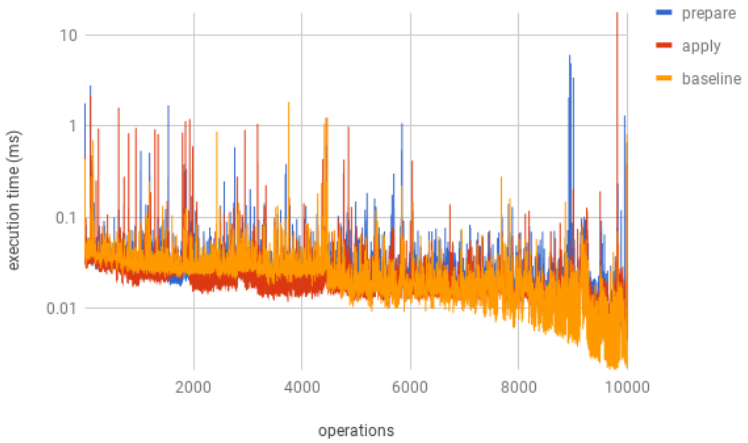


(a) Execution time

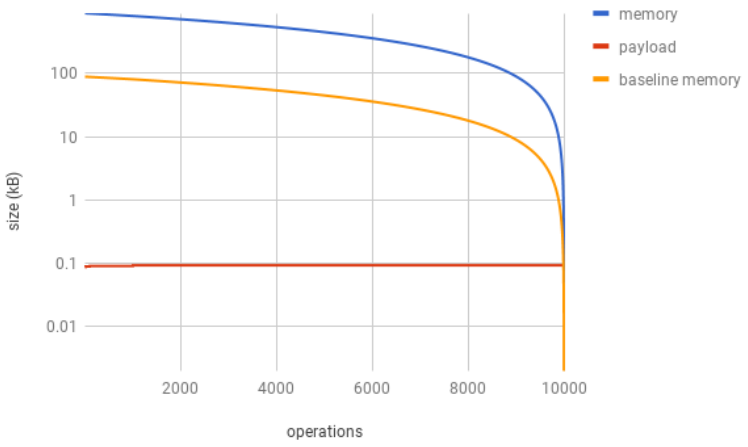


(b) Memory

Figure 4.10: Benchmark executing 10000 map add operations with unique keys.

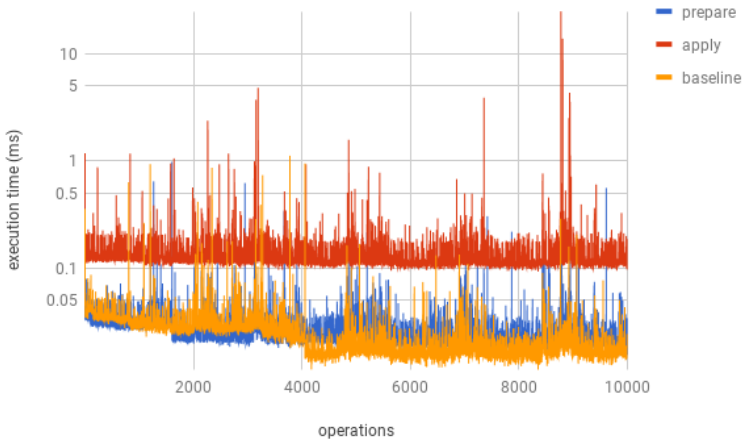


(a) Execution time

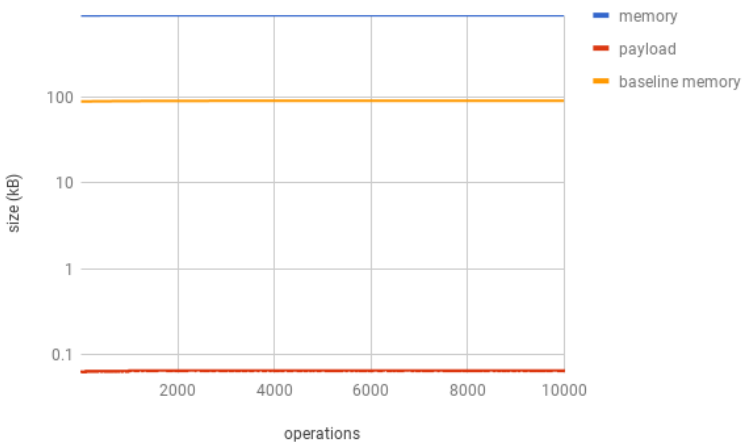


(b) Memory

Figure 4.11: Benchmark executing 10000 map remove operations from map.



(a) Execution time



(b) Memory

Figure 4.12: Benchmark executing 10000 register set operations.

Chapter 5

Discussion

The goal of this project was to implement a real-time collaborative data store. The requirements of this data store were formalized in section 2.4. Most of these requirements were met by the design of the data store, such as using CRDTs and thereby ensuring strong eventual consistency. All the data structures we wanted to support could be implemented using the CRDTs: List, Map and Register. Some modifications were made to the List and Map from the literature and an additional layer was built on top to support arbitrary nesting inside Lists and Maps.

The implemented data store has some similarities to the JSON CRDT [18], mostly in the type of data that can be stored. However, we made several improvements to the JSON CRDT to make our solution scale better. For example, we removed most use of tombstones, do not use vector clocks and there are some differences in collaborative behaviour. There is also no implementation or performance evaluation in the literature. The performance and scalability of the implemented data store were evaluated using a set of benchmarks. The results of the benchmarks largely corresponds to the complexity analysis. All implemented data types have some memory and computational overhead compared to the baseline. Importantly, the Map and Register scale very well with respect to both the size of the data structure and the number of operations. In contrast, the memory and execution time for the List does not scale as well since it grows with the number of operations and not only the number of elements stored. This is a fundamental problem found in a lot of the literature on CRDTs [30] [18] and can be mitigated using distributed garbage collection algorithms.

Still, the current implementation should be acceptable for most applications where list operations are not so frequent. The benchmarks also show that the data store is very efficient to replicate over the network which is due to the operation based approach.

The purpose of the data store is that it should be general enough to be able to add collaborative behaviour to any application. Here, the collaborative behaviour was evaluated using a set of example scenarios which describe how concurrent edits are handled. We show that the concurrent updates were handled in a predictable way in all of the scenarios with the exception of concurrent assignments to the same key. This was the only case where information was lost without removing something.

Knowing exactly how general the data store really is can only be determined by using it to develop collaborative applications. While the data format itself should be enough to encode most application state, the need for custom collaborative behaviour is unclear. One such custom feature implemented in this data store is the move operation which allows users to concurrently update and reorder list items. The move operation is unique to our implementation and is not described in any literature to our knowledge.

Collaboration, the act of working together towards a common goal, is an important part of society. Collaboration is one of the main use cases of the internet and can be used in all parts of society such as medicine, research, business, education etc. Doing collaboration in real-time can further increase the ease of use of such applications. The data store developed as a part of this project allows application developers to implement real-time collaborative applications without having to worry about replication, conflict resolution and consistency. This makes it a lot easier to develop and can increase the number of collaborative applications. Thus, use of the data store can be of great benefit to to the public.

Efficient collaboration may also have impact on economical sustainability since it can make more effective use of resources. Furthermore, being able to collaborate over the internet means that people can easier work together over large distances, decreasing the need for traveling. Hence, contributing to environmental sustainability by lowering the negative impact on the environment incurred by traveling.

Privacy and security is one of the most important ethical issues on the internet. CRDTs does not require a central server and can be used

peer-to-peer. This means that CRDTs can be used to develop secure decentralized applications which preserve privacy. The implemented data store can for example be used in applications to collaborate in a private and secure way.

There are a lot of things to consider for practical use of this technology. One such thing is security against malicious users. There is no validation for the operations being sent so a malicious or faulty user might send malformed data which might break the application. Ideally, there should be some form of schema for the data model, which could be evaluated on incoming data. Further research is needed on how to implement schema validation in a distributed setting.

Support for undo and redo of operations in the data store was not considered in this report and might require big changes to the design in order for it to work. This is something that should be investigated in the future.

The implemented data store is not proven to be consistent. However, all the underlying data structures are proven CRDTs. Thus, one could argue that the data store itself should be a CRDT as well. Another thing not thoroughly discussed in this report is the best way to implement reliable causal ordered message delivery, which is a prerequisite for the CRDT implementation to be consistent.

Improving the scalability of the List implementation can also be a source of future work.

Chapter 6

Conclusions

The objective of this thesis was to implement a general purpose data store that supports real-time collaborative editing.

The implemented data store uses building blocks of CRDTs previously defined in literature, which guarantees strong eventual consistency among replicas. Map, List and Register CRDTs are composed together in a new way to support collaborative editing of semi-structured data.

The data store supports concurrent Map add and remove, List insert, move and remove as well as Register set. Most of the concurrent updates are handled in a predictable way and merged so that no information is lost, with a few exceptions.

Using the CRDT based data store comes with a computational as well as memory penalty compared to using the corresponding non-CRDT data structures. The Map and Register implementation scales very well, but the List gets worse performance as the number of operations grows. However, replication over the network is very efficient.

The data store allows application developers to add collaborative behaviour to many types of applications without having to think about how the data is replicated.

Bibliography

[1] Mehdi Ahmed-Nacer et al. “Evaluating crdts for real-time document editing”. In: *Proceedings of the 11th ACM symposium on Document engineering*. ACM. 2011, pp. 103–112.

[2] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. “Interval tree clocks”. In: *International Conference On Principles Of Distributed Systems*. Springer. 2008, pp. 259–274.

[3] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. “Efficient state-based crdts by delta-mutation”. In: *International Conference on Networked Systems*. Springer. 2015, pp. 62–76.

[4] Peter Alvaro et al. “Consistency Analysis in Bloom: a CALM and Collected Approach.” In: *CIDR*. 2011, pp. 249–260.

[5] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. “Making operation-based CRDTs operation-based”. In: *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer. 2014, pp. 126–140.

[6] Annette Bieniusa et al. “An optimized conflict-free replicated set”. In: *arXiv preprint arXiv:1210.3368* (2012).

[7] Eric Brewer. “CAP twelve years later: How the" rules" have changed”. In: *Computer* 45.2 (2012), pp. 23–29.

[8] Eric A Brewer. “Towards robust distributed systems”. In: *PODC*. Vol. 7. 2000.

[9] Russell Brown et al. “Riak DT map: a composable, convergent replicated dictionary”. In: *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*. ACM. 2014, p. 1.

[10] Sebastian Burckhardt et al. “Cloud types for eventual consistency”. In: *European Conference on Object-Oriented Programming*. Springer. 2012, pp. 283–307.

[11] Giuseppe DeCandia et al. “Dynamo: amazon’s highly available key-value store”. In: *ACM SIGOPS operating systems review*. Vol. 41. 6. ACM. 2007, pp. 205–220.

[12] Clarence A Ellis and Simon J Gibbs. “Concurrency control in groupware systems”. In: *Acm Sigmod Record*. Vol. 18. 2. ACM. 1989, pp. 399–407.

[13] Seth Gilbert and Nancy Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[14] *Google Docs - create and edit documents online, for free.* `https://www.google.com/docs/about/`. (Accessed on 04/17/2018).

[15] *Google Drive Blog: What’s different about the new Google Docs: Conflict resolution*. `https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs_22.html`. (Accessed on 04/17/2018).

[16] *Interactive presentation software - Mentimeter*. `https://www.mentimeter.com/`. (Accessed on 04/17/2018).

[17] *jakobivarsson/neuron: Real-time collaborative data store*. `https://github.com/jakobivarsson/neuron`. (Accessed on 05/06/2018).

[18] Martin Kleppmann and Alastair R Beresford. “A Conflict-Free Replicated JSON Datatype”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (2017), pp. 2733–2746.

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

[20] Dahlia Malkhi and Doug Terry. “Concise version vectors in WinFS”. In: *International Symposium on Distributed Computing*. Springer. 2005, pp. 339–353.

[21] Friedemann Mattern et al. “Virtual time and global states of distributed systems”. In: *Parallel and Distributed Algorithms* 1.23 (1989), pp. 215–226.

[22] Gérald Oster et al. “Real time group editors without operational transformation”. PhD thesis. INRIA, 2005.

[23] D Stott Parker et al. “Detection of mutual inconsistency in distributed systems”. In: *IEEE transactions on Software Engineering* 3 (1983), pp. 240–247.

[24] 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.

[25] Nuno Preguiça et al. “Brief announcement: Efficient causality tracking in distributed storage systems with dotted version vectors”. In: *Proceedings of the 2012 ACM symposium on Principles of distributed computing*. ACM. 2012, pp. 335–336.

[26] *Read Me - Redux*. `https://redux.js.org/`. (Accessed on 06/04/2018).

[27] Hyun-Gul Roh et al. “Replicated abstract data types: Building blocks for collaborative applications”. In: *Journal of Parallel and Distributed Computing* 71.3 (2011), pp. 354–368.

[28] Yasushi Saito and Marc Shapiro. “Optimistic replication”. In: *ACM Computing Surveys (CSUR)* 37.1 (2005), pp. 42–81.

[29] Marc Shapiro et al. “A comprehensive study of convergent and commutative replicated data types”. PhD thesis. Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[30] Marc Shapiro et al. “Conflict-free replicated data types”. In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[31] Chengzheng Sun and Clarence Ellis. “Operational transformation in real-time group editors: issues, algorithms, and achievements”. In: *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. ACM. 1998, pp. 59–68.

[32] Werner Vogels. “Eventually consistent”. In: *Communications of the ACM* 52.1 (2009), pp. 40–44.

[33] Stéphane Weiss, Pascal Urso, and Pascal Molli. “Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks”. In: *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*. IEEE. 2009, pp. 404–412.

TRITA -EECS-EX-2019:28