Garbage Collected CRDTs on the Web

Studying the Memory Efficiency of CRDTs in a Web Context

Astrid Rehn

5th August 2021

Abstract

In today's connected society, where it is common to have several connected devices per capita, it is more important than ever that the data you need is omnipresent, *i.e.* its available when you need it, no matter where you are. We identify one key technology and platform that could be the future—peer-to-peer communication and the Web. Unfortunately, guaranteeing consistency and availability between users in a peer-to-peer network, where network partitions are bound to happen, can be a challenging problem to solve.

To solve these problems, we turned to a promising category of data types called CRDTs—Conflict Free Replicated Data Types. By following the scientific tradition of reproduction, we build upon previous research of a CRDT framework, and adjust it work in a peer-to-peer Web environment, *i.e.* it runs on a Web browser. CRDTs makes use of meta-data to ensure consistency, and it is imperative to remove this meta-data once it no longer has any use—if not, memory usage grows unboundedly making the CRDT impractical for real-world use. There are different garbage collection techniques that can be applied to remove this meta-data.

To investigate whether the CRDT framework and the different garbage collection techniques are suitable for the Web, we try to reproduce previous findings by running our implementation through a series of benchmarks. We test whether our implementation works correctly on the Web, as well as comparing the memory efficiency between different garbage collection techniques. In doing this, we also proved the correctness of one of these techniques.

The results from our experiments showed that the CRDT framework was well-adjusted to the Web environment and worked correctly. However, while we could observe similar behaviour between different garbage collection techniques as previous research, we achieved lower relative memory savings than expected. An additional insight was that for long-running systems that often reset its shared state, it might be more efficient to not apply any garbage collection technique at all.

There is still much work to be done to allow for omnipresent data on the Web, but we believe that this research contains two main takeaways. The first is that the general CRDT framework is well-suited for the Web and that it in practice might be more efficient to choose different garbage collection techniques, depending on your use-case. The second take-away is that by reproducing previous research, we can still advance the current state of the field and generate novel knowledge—indeed, by combining previous ideas in a novel environment, we are now one step closer to a future with omnipresent data.

Sammanfattning

I dagens samhälle är vi mer uppkopplade än någonsin. Tack vare det faktum att vi nu ofta har fler än en uppkopplad enhet per person, så är det viktigare än någonsin att ens data är tillgänglig på alla ens enheter—oavsett vart en befinner sig. Två tekniker som kan möjliggöra denna “allnärvaro” av data är Webben, alltså kod som körs på en Webbläsare, tillsammans med peer-to-peer-kommunikation; men att säkerställa att distribuerad data både är tillgänglig och likadan för alla enheter är svårt, speciellt när enhetens internetanslutning kan brytas när som helst.

Conflict-free replicated data-types (CRDT:er) är en lovande klass av datatyper som löser just dessa typer av problem i distribuerade system; genom att använda sig av meta-data, så kan CRDT:er fortsätta fungera trots att internetanslutningen brutits. Dessutom är de garanterade att konvergera till samma sluttillstånd när anslutningen upprättas igen. Däremot lider CRDT:er av ett speciellt problem—denna meta-data tar upp mycket minne trots att den inte har någon användning efter en stund. För att göra datatypen mer minneseffektiv så kan meta-datan rensas bort i en process som kallas för skräpsamling.

Vår idé var därför att reproducera tidigare forskning om ett ramverk för CRDT:er och försöka anpassa denna till att fungera på Webben. Vi reproducerar dessutom olika metoder för skräpsamling för att undersöka om de, för det första fungerar på Webben, och för det andra är lika effektiv i denna nya miljö som den tidigare forskningen pekar på.

Resultaten från våra experiment visade att CRDT-ramverket och dess olika skräpsamlingsmetoder kunde anpassas till att fungera på Webben. Däremot så noterade vi något högre relativ minnesanvändning än vad vi har förväntat oss, trots att beteendet i stort var detsamma som den tidigare forskningen. En ytterligare upptäckt vad att i vissa specifika fall så kan det vara mer effektivt att inte applicera någon skräpsamling alls.

Trots att det är mycket arbete kvar för att använder CRDT:er peer-to-peer på Webben för att möjliggöra “allnärvarande” data, så innehåller denna uppsats två huvudsakliga punkter. För det första så fungerar det att anpassa CRDT-ramverket och dess olika skräpsamlingsmetoder till Webben, men ibland är det faktiskt bättre att inte applicera någon skräpsamling alls. För det andra så visas vikten av att reproducera tidigare forskning—inte bara visar uppsatsen att tidigare CRDT-forskning kan appliceras i andra miljöer, dessutom kan ny kunskap hämtas ur en sådan reproducering.

Contents

List of Figures

Chapter 1

Introduction

In a peer-to-peer distributed system, consistency between replicated data among the peers can be difficult to achieve—latency and packet lost in the network can cause messages between peers to be delivered in an arbitrary order, resulting in a different final state of each replica [Shapiro et al., 2011]. There are different methods to resolve such conflicts, *e.g.* rolling back state, but this may be undesirable since it results in data loss. Conflict-Free Replicated Data Types (CRDTs) is a family of data types that can be used in a distributed peer-to-peer system to provide consistency among several replicas without the need for any synchronisation between them [Shapiro et al., 2011]. CRDTs achieves this by relying on commutativity, *i.e.* all replicas will converge to a common final state no matter the order the operations are applied.

CRDTs are considered to be a "solution" to the *Consistency, Availability and Partition Tolerance-theorem*, or the CAP-theorem. The CAP-theorem states that a distributed system can only provide two out the three following guarantees: consistency, availability, and partition tolerance [Butterfield et al., 2016]. To illustrate what this means in practice, imagine editing a document collaboratively with other peers; if a network partition occurs, *e.g.* you loose internet connection, we would have two network partitions. Without any means of communication between the partitions, you will either loose consistency, *i.e.* your respective document diverges into two different states, or you will have to sacrifice availability, *e.g.* by disallowing peers to write to the document. While CRDTs cannot guarantee consistency during a network partition, they help with conflict resolution when the partitions are re-joined; furthermore, by relying on peer-to-peer communication, there is no single-point of the failure and the system can therefore theoretically still function if you loose connection to some peers.

One possible application of peer-to-peer CRDTs is to realise a future envisioned by Baldemair et al. [2013], namely a future where data is available "anywhere and anytime to anyone and anything"—something we call omnipresent data. Baldemair et al. [2013] identified peer-to-peer communication to be a key part of reaching this future, a notion reinforced by the fact that there are more than five connected devices per capita in Western Europe today [Cisco Public, p. 7]—and the number are still increasing. If these devices can communicate peer-to-peer without requiring any conflict resolution, that future is now closer than ever.

Omnipresent data comes in many shapes, some that are user-facing while others are hidden from end-users. One example of a system that is hidden from end-users is Redis, an in-memory data structure store, that has built-in replication and provides high availability via their Redis Sentinel service[1] that has one primary replica, and multiple secondary replicas. If the secondary replicas together agree that the primary replica no longer is available, they will collectively elect a new primary to continue to remain available [redislabs, n.d.]; while this makes it more available, it can still not guarantee availability in the system as a whole if all replicas *e.g.* crash, but this trade off is worth it to keep consistency. On the other hand, there are end-user facing applications like Google Docs[2], a word processor that runs on the Web, *i.e.* in a Web browser, that provides near real-time collaboration between clients. In this scenario, consistency is not as important during editing, only the end result of the document is important—so, instead of sacrificing availability, consistency is sacrificed temporarily. There are many near-real time collaboration tools that run on the Web with the same properties[3][4][5][6] but none, at least that the author knows of, communicates peer-to-peer.

The World Wide Web Consortium—the community whose work is to develop open standards for the Web—cites two main principles of their mission, all of which is to ensure the long-term growth of the Web: the Web for All principle and the Web on Everything principle [w3c, 2017]. The Web for All principle recognises the social value of the Web and how it enables human communication to be benefits that should be available to all. On the other hand, the Web on Everything principle notes that the number of devices that can access the Web has grown and will continue to do so, thus, work must be done to ensure this remains true in the future. These two principles succinctly convey the importance of the Web as a platform—it is global and accessible to all, no matter what device you use, which results in the potential to reach more people than was ever possible previously.

One probable reason for the lack of peer-to-peer applications on the Web may be due to the historically limited set of Web APIs, the API surface area available to Web browsers. In fact, the only way to achieve peer-to-peer communication on the Web is via the WebRTC APIs [WebRTC Specification], a set of APIs whose specification did not transition from working draft status to candidate recommendation status until 2017 [WebRTC Recommendation].

1.1 Purpose and Goals

While CRDTs shows great promise, *e.g.* inching closer the future envisioned by Baldemair et al. [2013], it comes with its own suite of deficiencies. One of these is that many CRDTs have to store meta-data to provide its consistency guarantees, and if no schemes for removing meta-data are employed, memory use may grow without bound [Bauwens and Gonzalez Boix, 2019].

Previous work by Bauwens and Gonzalez Boix [2019] has proposed, implemented and evaluated a way to remove this meta-data—a process known as garbage collection—in the Lua programming language, in a peer-to-peer setting. They did

[1]`https://redis.io/`
[2]`https://www.google.com/docs/about/`
[3]`https://www.figma.com/`
[4]`https://www.office.com/`
[5]`https://glitch.com/`
[6]`https://trello.com/`

this by extending a general framework for building CRDTs, proposed by Baquero et al. [2017]. They then tested the effectiveness of this extension on a specific type of CRDT called the *Add-Wins Set*.

There would be a tangible benefit of having memory efficient CRDTs on the Web running over peer-to-peer for several reasons. Firstly, the rising importance of the Web creates a need to provide a consistent user experience across a plethora of heterogeneous devices, regardless of the device's limitations. Lastly, since many applications that can make use of CRDTs, *e.g.* real-time collaborative tools, are popular on the Web, there is a real-world need and use for CRDTs. Consequently, any improvements to the underlying data-structures would, as a direct consequence, improve the tools built on-top of them.

Lua, however, is unfortunately incompatible with the Web, not only because the Web runs a different programming language—JavaScript [Mozilla Developer Network, a] and more lately WebAssembly [Mozilla Developer Network, b]—but because they are limited to the Web APIs mentioned previously. Consequently, Bauwens' and Gonzalez Boix's [2019] work is difficult to apply directly in the Web environment.

The goal of this thesis is to reproduce the findings of Bauwens and Gonzalez Boix [2019], not by re-implementing the eager meta-data removal in Lua, but by designing and re-engineering the same method for meta-data removal on the Web. To achieve this goal, we had to

- **implement** the general CRDT framework by Baquero et al. [2017], in a peer-to-peer setting on the Web, *i.e.* running on a Web browser, thereafter we
- **implement** the garbage collection method proposed by Bauwens and Gonzalez Boix [2019], and at last we
- **evaluated** our implementation to see if (1) the general CRDT framework can at all be adapted to run on the Web over a peer-to-peer network and (2) if similar improvements to memory usage can be seen in the Web environment while applying the garbage collection technique proposed by Bauwens and Gonzalez Boix [2019].

Chapter 2

Background

As mentioned in § 1.1, conflict-free replicated data-types (CRDTs) are a useful construct to design highly available and consistent distributed systems, even in the event of partitioned networks [Shapiro et al., 2011]. This chapter describes the necessary theory needed to achieve our goal of implementing a memory efficient, peer-to-peer, CRDT framework for the Web.

We start by discussing the problems of having highly available data in a peer-to-peer environment, and how CRDTs tries to tackle this problem, in § 2.1 and § 2.2 respectively. After this, we introduce the necessary building blocks for designing CRDTs in § 2.3 and § 2.4, such as causal ordering and ways to broadcast data to many peers in a network. This is later followed by § 2.5 that both introduces the general CRDT framework proposed by Baquero et al. [2017] and mentions the shortcomings of this design—different solutions to these problems are presented in § 2.6, including the eager garbage collection proposed by Bauwens and Gonzalez Boix [2019]. Finally, in § 2.7, we investigate ways to set up a peer-to-peer network on the Web.

2.1 Availability of Replicated Data

Data replication in a distributed system is the act of storing the same data in multiple locations simultaneously, where each copy is called a replica, often used to achieve high performance and availability [Vogels, 2008]. Data replication is not a perfect solution, and comes with its own set of consequences, mainly those defined by the CAP-theorem. The CAP-theorem states that a distributed system can only provide a maximum of two out of the three following guarantees: consistency, availability and partition tolerance [Butterfield et al., 2016]. For example, a partitioned network cannot achieve consistency between the partitions unless it sacrifices availability, *e.g.* by disallowing writes.

When partitions do occur, *e.g.* a mobile user travelling through a tunnel with no signal, the CAP-theorem forces the system to either sacrifice its availability or consistency. If the system sacrifices availability, the mobile user would be blocked to perform any work. On the other hand, if the consistency guarantee is broken, the mobile user would be able to continue work but it would not receive new updates from other users, nor would they receive updates from the mobile user, causing the different user's replicas to diverge. This divergence may be acceptable, if the system is able to converge again once the network partitions are joined, which is the idea behind the consistency model called *eventual consistency* [Vogels, 2008]. Eventual consistency (EC) is a consistency model that guarantees that if no writes are made to a replica, then subsequent reads will eventually return the latest update [Vogels, 2008]. An example of a popular system that employs EC is the Doman Name System (DNS). Just as with DNS, EC gives no promises as of when consistency is achieved, but predictions can be made for a specific system given some parameters, *e.g.* latency, the number of replicas in the system and cache expiration delays [Vogels, 2008].

A system can guarantee EC in several ways, *e.g.* by detecting and rolling back the conflicting updates, a brittle and error-prone approach with little guidance from literature [Shapiro et al., 2011]. Strong eventual consistency (SEC) is a consistency model which imposes further restrictions on the definition of EC; a system is considered to be SEC if it guarantees both EC and that all "correct replicas that have delivered the same updates have equivalent state" [Shapiro et al., 2011]. Conflict-free replicated data types (CRDTs) is a category of data types that are natively SEC by leveraging the mathematical property of commutativity [Shapiro et al., 2011].

2.2 Conflict-Free Replicated Data Types (CRDTs)

Conflict-Free Replicated Data Types (CRDTs) are data types that can guarantee SEC in a distributed setting without having to resort to conflict resolution, synchronisation or roll-back, by relying on the property of commutativity where possible [Shapiro et al., 2011]. Conversely, for operations that are not natively commutative, commutativity can be manufactured by embedding ordering information in the operation data and applying operations in this specific order [Baquero et al., 2017, p. 7]. These techniques will be further described in § 2.5.

An example of a simple, commutative, CRDT is a numeric counter. A counter has two operations: increment and decrement. Both these operations are commutative, *i.e.* $+1 - 1 = -1 + 1$, and so, a counter would remain available and SEC, even in a partitioned network [Shapiro et al., 2011], given that the network is joined together sometime in the future so that operations can be delivered.

The literature differentiates between state-based and operation-based CRDTs [Shapiro et al., 2011], where the first would resolve a final state from two diverged states based on a merge function. Operation-based CRDTs, on the other hand, reaches a convergent final state based on only operations applied to a replica. For the rest of this thesis, CRDT will refer to the operation-based type.

Baquero et al. [2017] defines a system model in which they later define a method to design operation-based CRDTs in; the distributed system consists of N number of *nodes*, each assigned with a globally unique identifier $i \in 1 \dots N$. All nodes communicate asynchronously via message passing at different speeds, and messages—similar to the internet—can be lost, reordered or duplicated. The system experiences arbitrary, but transient, network partitions and nodes may crash, but are assumed to eventually recover and to have an intact state. Each node stores a replica of a shared data type, initially all equivalent to each other. When a node applies an operation to its local replica, it will be broadcast to all other nodes via a reliable causal broadcast, a broadcasting middleware which ensures exactly-once, reliable and causally ordered delivery of messages—the reliable causal broadcast will be further expanded upon in § 2.3 and § 2.4.

$\sigma_i \in \Sigma$	State space, of which σ_i is an instance of at node i
$\mathsf{prepare}_i(\mathsf{operation}, \sigma_i)$	Prepares a message for the given operation, potentially embedding data that depends not only on the operation, but on the current state of the CRDT
$\mathsf{effect}_i(\mathsf{message}, \sigma_i)$	Mutates the state given a message
$\mathsf{eval}_i(\mathsf{query}, \sigma_i)$	Read-only evaluation of a query on the state

Figure 2.1: General scheme for designing operation-based CRDTs[Baquero et al., 2017, Figure 1].

In the system model described above, Baquero et al. [2017] defines a general scheme for operation-based CRDTs, which is found in Fig. 2.1. The scheme consists of three functions—prepare, effect and eval—over a state space Σ. To implement an increment-only counter in this scheme, *i.e.* a counter that only supports the increment operation, the state $\sigma_i \in \Sigma$ for node i would be

$$\sigma_i \in \Sigma \equiv \mathbb{N}$$

and the only operation, increment, would prepare the following message m which would cause the following effect

$$\begin{aligned} \mathsf{prepare}_i(\mathsf{increment}, \sigma_i) &= m = \mathsf{increment} \\ \mathsf{effect}_i(m, \sigma_i) &= \sigma_i + 1. \end{aligned}$$

Note that for this specific data-type, prepare does not need to embed any extra data, so the message m is simply the same as the operation itself. At last, we have

$$\mathsf{eval}(\mathsf{value}, \sigma_i) = \sigma_i$$

to evaluate the value of the counter, which in this case, simply returns the value of the stored counter.

Fig. 2.2 shows the increment-only counter in practice, where the counter implemented in this general CRDT scheme is operated between two nodes, A and B. In the figure, node A first prepares a message with the preparefunction, applies it to it's local replica with the effectfunction and later sends this over to node B. Node B receives this message, and applied it to it's own local replica. At this point, both A's and B's replicas evaluate to 1. After this, node B does the exact same thing, and sends the message over to node A, who in turn, applies it to it's own local replica. At the end, we note that both node A's and node B's state are equivalent and SEC is thus uphold.

Baquero et al. [2017] defines that, if a CRDTs prepared messages contain only the contents of the operation itself, they are called *pure*. The increment-only counter is then be pure since

$$\mathsf{prepare}(\mathsf{operation}, \sigma) = \mathsf{operation}$$

and this is a property not all CRDTs natively have. Pure CRDTs are desirable since their design is trivial compared to that of non-pure CRDTs [Baquero et al., 2017, p. 5]. Consider the semantics of the *Add-Wins Set*—similar to a regular set it has an add and a remove operation, but in the event of a concurrent add and remove, the add takes precedence. This is illustrated in Fig. 2.3, where two nodes, A and B, creates two

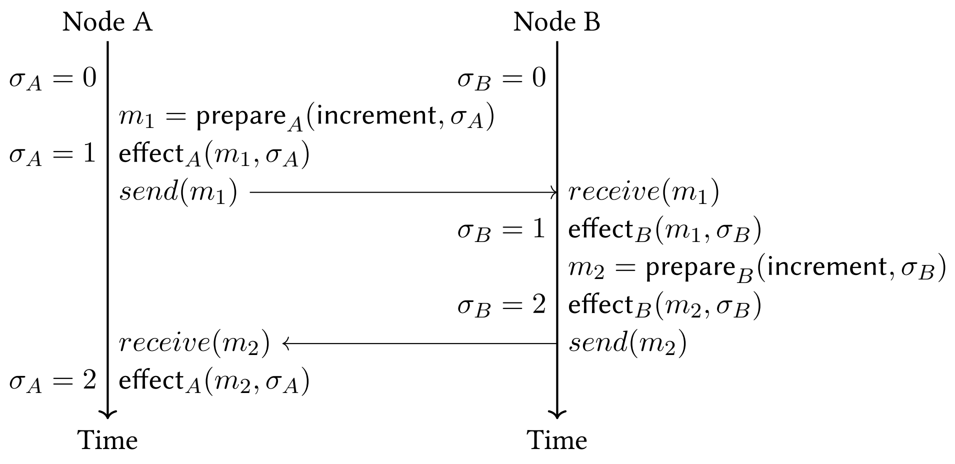

Node A
Node B
$\sigma_A = 0$
$\sigma_B = 0$
$m_1 = \mathsf{prepare}_A(\mathsf{increment}, \sigma_A)$
$\sigma_A = 1$
$\mathsf{effect}_A(m_1, \sigma_A)$
$send(m_1)$
$receive(m_1)$
$\sigma_B = 1$
$\mathsf{effect}_B(m_1, \sigma_B)$
$m_2 = \mathsf{prepare}_B(\mathsf{increment}, \sigma_B)$
$\sigma_B = 2$
$\mathsf{effect}_B(m_2, \sigma_B)$
$receive(m_2)$
$send(m_2)$
$\sigma_A = 2$
$\mathsf{effect}_A(m_2, \sigma_A)$
Time
Time

Figure 2.2: Two nodes, A and B, operates an increment-only counter by following the CRDT scheme in Fig. 2.1. The evaluated state is shown to the left of each timeline, while the right-hand side of each timeline is the functions called by each node, from top to bottom. The arrows between A's and B's timelines illustrates sending a message over the network.

concurrent operations of the same element, number 1. After the operation has been applied locally, and before the remote operation has been applied, the nodes evaluate their respective state to inspect the elements in the set: note that the state differs for A and B. Indeed, according to SEC, the state is allowed to diverge. However, once the operation has been delivered to the respective node, since add "wins", both nodes will evaluate to see that the number one is indeed in the set—all nodes now have a consistent state and satisfies SEC.

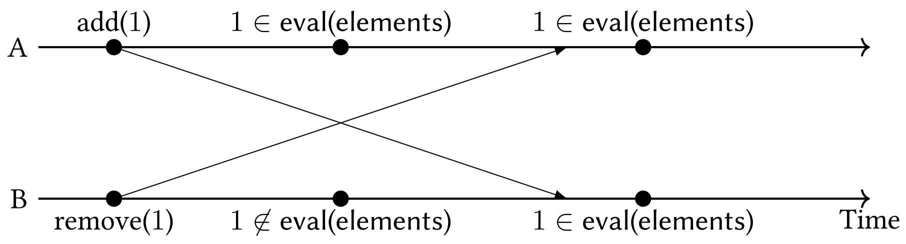


Figure 2.3: The nodes A and B sends a concurrent add and remove operation of the same element to each other. In the end, however, the "add" prevails and both replicas has a consistent state

To model the operation-based scheme in Fig. 2.1 after the semantics of the Add-Wins Set, it requires embedding meta-data in the prepare function in order to decide an ordering between operations, and identify concurrent operations [Baquero et al., 2017, p. 5–6]. This meta-data is not part of the operation itself, which makes it non-pure. The ordering used for this is a partial order called *causal ordering*.

2.3 Ordering Operations

As mentioned in § 2.2, the system model described uses a reliable causal broadcast. One property of the reliable causal broadcast is that it delivers operations in a *causal* *ordering*. The causal order is useful due to the fact that no two computer clocks can be perfectly synchronised across a distributed system, which prevents the system being able to rely on physical time to order operations between different nodes [Coulouris et al., 2012, p. 607]. The causal ordering relation between two events e and e' is denoted by $e \rightarrow e'$ and means that e is causally ordered before e'. The causal ordering relation is established by three criteria [Coulouris et al., 2012, p. 607]

1. for any message m, we have that sending m from a sender is causally ordered before receiving the message at the recipient r *i.e.* $send(m) \rightarrow receive_r(m)$
2. if events e occurred before e' at node i, then $e \rightarrow e'$.
3. given that events $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$, also known as transitivity over $\rightarrow$.

Thus, if any of the three criteria above holds, an event can be said to be causally ordered before the other. If none of the above criteria hold—i.e. $e \not\rightarrow e'$ and $e' \not\rightarrow e$—then e and e' is said to be concurrent [Coulouris et al., 2012, p. 608]. Figure Fig. 2.4 demonstrates a scenario where two nodes, A and B, passes messages m_1, m_2 and m_3 between each other over the course of three steps, (a), (b) and (c). Some of the facts that can be concluded by looking at the figure and following the above criteria are:

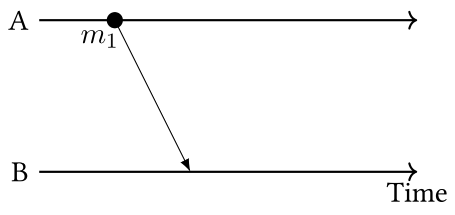
- In Fig. 2.4a, together with the first criteria, we can see that $send(m_1) \rightarrow receive_B(m_1)$
- Similarly in Fig. 2.4b we have that $send(m_2) \rightarrow receive_A(m_2)$
- Furthermore, in Fig. 2.4b, together with the second criteria, we have that $receive_B(m_1) \rightarrow send(m_2)$.
- Additionally, the third criteria then yields $send(m_1) \rightarrow receive_B(m_1) \rightarrow send(m_2) \rightarrow receive_A(m_2)$ through the transitive property.
- Finally, in Fig. 2.4c, we see that none of the criteria can establish a relation between $send(m_2)$ and $send(m_3)$, they are thus concurrent.

To ensure causal delivery of messages, causal ordering information has to be embedded into the messages sent between nodes. *Lamport timestamps* is one method of representing the causal ordering. Each node i stores a counter L_i which it uses to timestamp events. The counter is incremented by the following rules [Coulouris et al., 2012, p. 608]

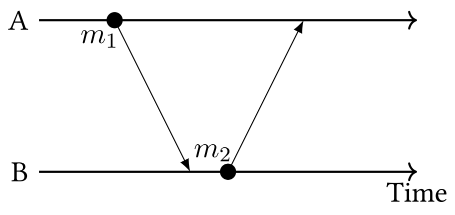
1. L_i is incremented by one before each event issued at node i, i.e. $L_i = L_i + 1$
2. (a) When node i sends a message m, we attach the current timestamp value to this message, $t = L_i$.
   (b) When a node $j \neq i$ receives the message (m, t), the node computes $L_j = max(L_j, t)$.

This scheme captures the fact that if $e \rightarrow e'$ then $L(e) < L(e')$, where $L(e)$ is the timestamp of event e [Coulouris et al., 2012, p. 608]. However, the reverse is not true, *i.e.* $L(e) < L(e') \not\Rightarrow e \rightarrow e'$ [Coulouris et al., 2012, p. 608], and consequently, it is not enough to piggyback Lamport timestamps on to messages to achieve a causal ordering of delivered messages.

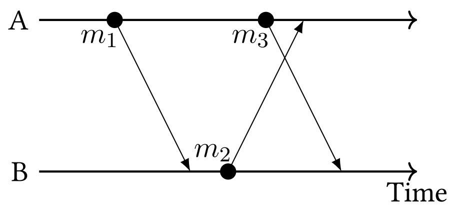
To overcome Lamport timestamps deficiency, *vector clocks* can be used, which is a generalisation of Lamport timestamps that gives more information about the system state. A vector clock, for a system with N nodes is an array of N counters. Just as with the Lamport timestamps, each node i timestamps events with its own vector



(a) Node A sends m_1 to node B.



(b) Node B sends m_2 to node A.



(c) Before node A receives m_2, it sends m_3 to node B.

Figure 2.4: Two nodes, A and B, sending messages to each other in three steps, (a), (b) and (c). A filled ball at a timeline indicates the creation and sending of message m_n, the arrow tip indicates receiving the message at the node corresponding to the timeline.

clock V_i. The process is similar to that of Lamport timestamps [Coulouris et al., 2012, p. 609]:

1. Just before node i issues an event, it sets $V_i[i] = V_i[i] + 1$
2. (a) When node i sends a message m, we attach the current state of the vector clock V_i to this message
   (b) When a node $j \neq i$ receives the message (m, V_i), the node computes the component-wise maximum of both its own vector clock, and the received vector clock, $V_j[n] = max(V_j[n], V_i[n]), n \in [1, N]$.

To decide the causal ordering between messages, vector clock values can be compared as follows [Coulouris et al., 2012, p. 609]

$$\begin{aligned} V = V' &\text{ iff } V[n] = V'[n], n \in [1, N] \\ V \leq V' &\text{ iff } V[n] \leq V'[n], n \in [1, N] \\ V < V' &\text{ iff } V \leq V' \wedge V \neq V'. \end{aligned}$$

if $V(e)$ denotes the vector timestamp of event e, it is the case that $e \to e'$, then $V(e) < V(e')$. This allows a node to decide whether two events are causally ordered, based solely on the vector timestamps [Coulouris et al., 2012, p. 610]. Furthermore, concurrent events can be identified when $V(e) \nleq V(e')$ and $V(e') \nleq V(e)$, which we can more succinctly write as $V(e) \parallel V(e')$ [Coulouris et al., 2012, p. 610].

To demonstrate the differences between Lamport timestamps and vector clocks, a scenario is shown in Fig. 2.5. This figure shows two nodes, A and B, communicating with each other. Each message contains both a Lamport timestamp and a vector clock, initially set to zero. Messages m_1, m_2 and m_3 demonstrates clearly why vector clocks is needed to decide the causal ordering. We have that $m_1 \to m_3$, which both the Lamport timestamp and vector clock agrees on, *i.e.* $L_1 < L_3$ and

$V_1 < V_3$. We then know that m_1 and m_3 are both concurrent to m_2, which the vector clocks accurately reflects, *i.e.* $V_1 \parallel V_2$ and $V_2 \parallel V_3$. The Lamport timestamps, however, does not reflect this fact—while $L_1 = L_2$, we have that $L_2 < L_3$, despite the fact that m_2 and m_3 are concurrent.

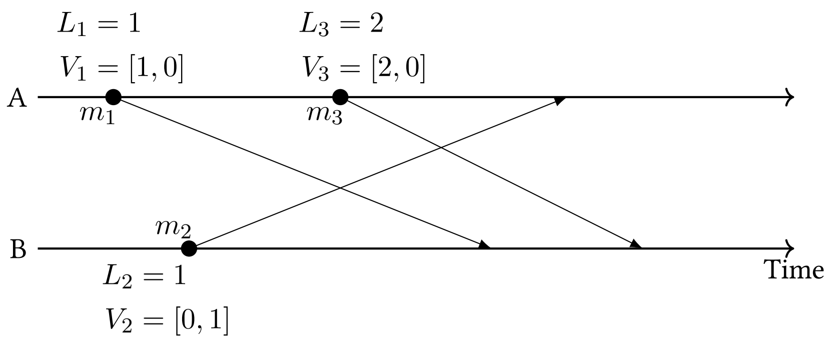


Figure 2.5: Illustration of differences between Lamport timestamps attached to the messages, written as L_n, versus vector clocks attached to the messages V_n. The subscripts of the timestamps/clocks correspond to the subscript of the message that it is attached to.

Now that the vector clock is defined, we can use the information embedded in them to find the causal ordering of the events from all nodes. If an ordering has been decided, an abstraction can be created so that all messages passed between nodes are delivered in the relevant order. This abstraction is called the causal ordered broadcast and it is the basis for communication within the CRDT network, together with other broadcasting variants.

2.4 Broadcasting in a Network

When communicating within several nodes in a network, we have some tools to our disposal. The IP protocol supports multicasting, *i.e.* sending the same message to multiple recipients, but unfortunately, it does not offer any validity or integrity guarantees—messages may be delivered corrupt, out of order, or not at all [Coulouris et al., 2012, p. 647]. Instead, a basic multicast can be designed using a reliable one-to-one communication channel, *e.g.* TCP [Coulouris et al., 2012, p. 647]. The basic multicast defines two primitives, B-multicast and B-deliver which is built on top of an underlying communication channel's $send_s^r$ and $receive_s^r$ functions—which is read as sending and receiving from sender s to recipient r—and looks like the following

```
**on** B-multicast($g$, $m$) at node $s$:
    $\forall r \in g : send_s^r(m)$

**on** $receive_s^r(m)$ at node $r$:
    B-deliver($m$)
```

where the primitive B-multicast sends the message m to each member node r in the group g [Coulouris et al., 2012, p. 647]. Once a message has been received at a node—via the underlying communication channel's receive function—it can simply be delivered to the next layer in the middleware stack via the B-deliver primitive [Coulouris et al., 2012, p. 647]. To further simplify, let's define the *basic broadcast*

(BB), and its corresponding primitive B-broadcast(m) to denote multicasting to all nodes connected in any relevant network, *e.g.* a network with all nodes who have a replica of a CRDT.

In the BB scheme, messages can be delivered in different orders at different nodes, due to the lack of any ordering guarantees. It can be desirable to deliver messages in the causal order, described in § 2.3. The causal broadcast (CB) consists of the primitives C-broadcast(m) and C-deliver(m), and is built on-top of the BB. Causal ordering can be ensured if each node i manages its own vector clock, V_i, as follows [Coulouris et al., 2012, p. 657]

```
**on** C-broadcast($m$) from node $i$:
    $V_i[i] = V_i[i] + 1$
    B-broadcast(<$V_i, m$>)

**on** B-deliver(<$V_i, m$>) at node $j \neq i$:
    place <$V_i, m$> in hold-back queue
    wait until $V_i[i] = V_j[i] + 1 \wedge V_i[k] \leq V_j[k]$ for $k \neq i$
    remove <$V_i, m$> from hold-back queue
    C-deliver($m$)
    $V_j[i] = V_j[i] + 1$
```

which in many regards relies on the vector clock algorithm described in § 2.3. One notable difference to the aforementioned algorithm is that instead of merging the two vector clocks—*i.e.* by taking the component-wise maximum—the algorithm increases the ith element by one. This is a simplification that can be made thanks to the condition in the deliver function, which implies that only the ith element, and no other, will increase by one [Coulouris et al., 2012, p. 657].

As mentioned in § 2.2, the system model used by Baquero et al. [2017] required a reliable causal broadcast (RCB). A reliable broadcast ensures that if any correct node in the network delivers a message, all other correct nodes will deliver the message as well [Coulouris et al., 2012]. The CB, which is built on-top of BB, does not satisfy this requirement, since the sender can fail midway through the broadcast, resulting in some nodes receiving the message, while some do not, resulting in an inconsistent state [Coulouris et al., 2012], thus violating the SEC.

To make the already defined CB reliable, the system has to ensure that a message is delivered to either all, or none, of the nodes. One approach is to make each node re-broadcast the message to all its peers once it delivers a message; each node keeps tracks of what messages it has delivered to make sure it only delivers every message once, even if it receives the message many times [Coulouris et al., 2012]. While simple, this approach is generally not considered practical, as every message is received at each node N times in a network of N nodes [Coulouris et al., 2012]. In the system model mentioned in § 2.2, both the BB, and to an extension, the CB, are reliable broadcasts, since node failures are assumed to be transient, *i.e.* a crashed node will eventually recover with its state intact. With the RCB defined, we can use this middleware to design pure CRDTs.

2.5 Designing Pure CRDTs

When designing CRDTs, Baquero et al. [2017] defines—and distinguishes between—two different types of CRDTs: commutative and non-commutative. A commutative data type is one where

1. for any two operations, any order of execution yields the same final state, and
2. all possible interleavings of concurrent operations result in a state equivalent to some linearisation of the operations, *i.e.* equivalent to one possible final state that would arise if the operations were not concurrent.

Both these requirements hold true for the increment-only counter described in § 2.2: the order of any two operations do not matter since the end state will result in a counter with a value of $+2$, and all possible interleavings of concurrent operations will be equivalent to some, in this case any, linearisation of the operations. For a set, conversely, this is not true since the order of your insertions and removals have an effect on the end state. For example, given the set S we have that

$$(S \cup \{1\}) \setminus \{1\} \neq (S \setminus \{1\}) \cup \{1\}$$

since on the left hand side we would have $1 \notin S$ while on the right hand side we would have $1 \in S$. Since the order of these two operations results in two different final state, it violates the first criteria and is thus not commutative [Baquero et al., 2017, p. 7].

Commutative data types can natively be designed as a pure CRDTs as defined in § 2.2—given that it is built on top of a reliable causal broadcast—and so shares their benefit of trivial design [Baquero et al., 2017, p. 6]. Many useful data types, however, are not commutative, *e.g.* the set mentioned above. This makes it difficult to represent it in a pure operation-based design since causal meta-data must be embedded within the operations to achieve commutativity, which complicates its design. Even worse, Baquero et al. [2017] note that the causal meta-data embedded in many CRDTs is duplicated in the underlying broadcasting middleware used, *e.g.* the *C-broadcast* mentioned in § 2.4. This makes the messages both more complex and increases the bandwidth usage, causing CRDTs to both be more difficult design and less efficient, respectively [Baquero et al., 2017, p. 2].

Baquero et al. [2017] suggest a general framework to design pure-operation based CRDTs for many different data types based on a partially ordered log, or PO-Log. First, by extending the CB middleware defined in § 2.4 to also expose the causal ordering information to upper layers, CRDTs can leverage this information in their design, instead of duplicating the data itself [Baquero et al., 2017, p. 7]. We have named this extended middleware *Tagged Causal Broadcast (TCB)*. Secondly, the general operation-based scheme for CRDTs, as shown in § 2.2, is extended by splitting it into two parts: one which is general for all CRDTs based on a Partially Ordered Log (PO-Log) and one which is data type-dependent. The idea is that by sharing code in a common CRDT framework, it partly decreases the implementation burden on developers, but also decreases the risk of introducing bugs, since more code exists in a well-tested common framework [Baquero et al., 2017, p. 9–10].

2.5.1 Tagged Causal Broadcast (TCB)

The tagged causal broadcast (TCB) defined by Baquero et al. [2017], is an extension to the CB described in § 2.4 that exposes causal ordering information to the upper layers. Similar to the CB, the TCB builds on the previously defined primitives B-broadcast and B-deliver, and defines the primitives TC-broadcast and TC-deliver as follows

```
on TC-broadcast($m$) from node $i$:
    $V_i[i] = V_i[i] + 1$
    B-broadcast(<$V_i, m$>)

on B-deliver(<$V_i, m$>) at node $j \neq i$:
    place <$V_i, m$> in hold-back queue
    wait until $V_i[i] = V_j[i] + 1 \wedge V_i[k] \leq V_j[k]$ for $k \neq i$
    remove <$V_i, m$> from hold-back queue
    TC-deliver(<$V_i, m$>)
    $V_j[i] = V_j[i] + 1$
```

meaning that the upper layer will receive the corresponding timestamp belonging to each message. Note that except for the line *TC-deliver(<V_i, m>)*, the TCB is identical to the normal CB.

There are two main benefits of exposing the causal information to the application layer when designing CRDTs. Since it avoids duplication of causality information in the application data logic, it (1) avoids the complexity of embedding causal information and (2) simplifies and reduces the message payload [Baquero et al., 2017, p. 2,10].

2.5.2 The PO-Log CRDT Framework

Baquero et al. [2017] suggests a general framework for designing pure operation-based CRDTs. The idea is to extend the general scheme for operation-based CRDT in Fig. 2.1 to make Σ, prepare and effect shared among all types of CRDTs, and deferring data type-specific semantics to the eval function. This is achieved with the help of a partially ordered log (PO-Log). The PO-Log stores all operations together with the operation's corresponding vector clock timestamp, which is fed to the CRDT framework by the TCB middleware described in § 2.5.1.

A reference design for pure operation-based designs based on the PO-Log is shown in Fig. 2.6. The state space, Σ is the PO-Log itself, and is represented by a map from timestamps to operations: $\Sigma = T \hookrightarrow O$ [Baquero et al., 2017, p. 10]. Initially, the state of the CRDT is set to an empty map: $\sigma_i^0 = \{\}$. Notice the simplicity of the prepare and effect, a side effect thanks to the design being pure, and deferring data type-specific semantics to the eval function, instead of effect. This means that effect can simply add the operation to the PO-Log, together with the timestamp supplied from the TCB middleware. Furthermore, Σ, prepare and effect are not tied to the semantics of any special data type, and as a result, it can be re-used for different data types implemented in this framework [Baquero et al., 2017, p. 10].

$$
\begin{array}{rcl}
\Sigma = T \hookrightarrow O & & \sigma_i^0 = \{\} \\
\mathsf{prepare}_i(o, \sigma_i) & = & o \\
\mathsf{effect}_i(o, t, \sigma_i) & = & \sigma_i \cup \{(t, o)\} \\
\mathsf{eval}_i(q, \sigma_i) & = & [\text{datatype-specific query function}]
\end{array}
$$

Figure 2.6: Reference design of a pure operation-based CRDT design based on a PO-Log [Baquero et al., 2017, Figure 4]. σ_i^0 defines the CRDTs initial state.

An Add-Wins Set, a set where an add is prioritised over a concurrent remove, can succinctly be described in this framework. Since Σ, prepare and effect are all data-type independent, only eval has to be designed. Baquero et al. [2017] gives a definition of eval for the Add-Wins Set, with operation o being either $[\mathsf{add}, v]$ or $[\mathsf{remove}, v]$. Then, for retrieving the elements in the set, eval is expressed as

$$\mathsf{eval}_i(\mathsf{elements}, \sigma_i) = \{v \mid (t, [\mathsf{add}, v]) \in \sigma_i \ \wedge (\nexists (t', [\mathsf{remove}, v]) \in \sigma_i,\ t < t')\}$$

which can be read as retrieving all values v that are of the type add in the PO-Log, given that there exists no remove of the same value v in the PO-Log, which are causally ordered after the aforementioned add operation.

This design, however, is not feasible for real-world use since the PO-Log grows linearly with the number of operations [Baquero et al., 2017, p. 10]. Fortunately, the PO-Log can be made more compact by identifying and removing garbage in the PO-Log, for example, if there are many adds of the same value, only one needs to be kept in the PO-Log. This will result in a structure that is more efficient, both computationally and with respect to memory.

2.6 Garbage Collection in CRDTs

The framework to design CRDTs using the PO-Log, explained in § 2.2, is unfeasible for real-world use, as the PO-Log grows unboundedly [Baquero et al., 2017, p. 10]. This leads not only to CRDTs that are computationally inefficient, but is wasteful of memory as well.

To tackle this problem, Baquero et al. [2017] proposes two PO-Log compaction techniques, which can be used in conjunction with each other, to trim the PO-Log of redundant operations and remove unnecessary meta-data. These two techniques, called *causal redundancy* and *causal stabilisation*, relies on the semantics of the data type itself and on yet another extension to the broadcasting middleware, respectively.

Finally, Bauwens and Gonzalez Boix [2019] proposes a method to speed up the PO-Log compaction of the causal stabilisation mechanism. It does this by eagerly collecting causality information and sharing this to other nodes in the network. The idea is that bandwidth can be traded for a more eager garbage collection, thus improving memory efficiency.

2.6.1 Causal Redundancy

Causal Redundancy is a mechanism that prunes the PO-Log in the effect step of the CRDT framework seen in Fig. 2.6. The general idea is to identify which operations are redundant, and if they are, they can be removed. The goal is to keep the least number of operations that still evaluate to the same final state as if the PO-Log had not been trimmed at all [Baquero et al., 2017, p. 12].

To support this, Baquero et al. [2017] extend the generic CRDT framework described in § 2.2, to the framework seen in Fig. 2.7. The additions to the framework is a more complex effect function which makes use of three data-type specific the redundancy relations: R, R_0 and R_1. The R relation defines whether the delivered operation is redundant by itself [Baquero et al., 2017, p. 12], e.g. a reset operation is redundant to store in the PO-Log since all older operations in the PO-Log can be removed instead. The removal of operations currently in the PO-Log is specified by the R_0 and R_1 relations; this decision may for some data types be dependent on whether a delivered operation is itself discarded or not [Baquero et al., 2017, p. 12–13]. Thus, R_0 is used when the delivered operation itself is discarded, while R_1 is used if the new arrival is kept in the PO-Log [Baquero et al., 2017, p. 12–13]. For many data types, these two relations are equal, and is then defined as $\mathsf{R}\_ = \mathsf{R}_0 = \mathsf{R}_1$ to denote both relations.

$$\begin{array}{rcl}
\Sigma = T \hookrightarrow O & & \sigma_i^0 = \{\} \\
\mathsf{prepare}_i(o, \sigma_i) & = & o \\
\mathsf{effect}_i(o, t, \sigma_i) & = & \{(t,o) \mid (t,o) \not\mathsf{R} s\} \cup \{x \in s \mid x \not\mathsf{R}_0 (t,o) \wedge \\
 & & (t,o) \mathsf{R} s \vee x \not\mathsf{R}_1 (t,o) \wedge (t,o) \not\mathsf{R} s\} \\
\mathsf{R}, \mathsf{R}_0, \mathsf{R}_1 & = & [\text{datatype-specific redundancy relations}] \\
\mathsf{eval}_i(q, \sigma_i) & = & [\text{datatype-specific query function}]
\end{array}$$

Figure 2.7: Reference design of a pure operation-based CRDT design based on a PO-Log with redundancy relations R, R_0 and R_1 [Baquero et al., 2017, Figure 7].

Fig. 2.8 shows an implementation of the Add-Wins Set in this extended framework. Since all parts, except the redundancy relations and eval function is data-type independent, only these are shown. This time, a clear operation has been added to show a more complex use case. In the Add-Wins Set, the aim is to only store adds in the PO-Log—clear operations and remove operations can simply remove old add operations from the PO-Log. To accomplish this, we define R to specify that all clear and remove operations are redundant, which means they will be discarded and thus not added to the PO-Log. The R_ is defined to specify that operations in the PO-Log, which are causally ordered before the new operation, are redundant, if the new operation is a clear operation, or the operation adds or removes the same value, i.e. has the same argument. At last, since only the latest add is found in the PO-Log—of which there have been no remove or clear operation afterwards—the eval function can simply return all values in the PO-Log.

$$\begin{array}{rcl}
(t, o)\ \mathsf{R}\ \sigma_i & \iff & o[0] = \mathsf{clear} \vee \mathsf{remove} \\
(t', o')\ \mathsf{R}\_\ (t, o) & \iff & t' < t \wedge (o[0] = \mathsf{clear} \vee o[1] = o'[1]) \\
\mathsf{eval}_i(\mathsf{elements}, \sigma_i) & = & \{\ v \mid (\_, [\mathsf{add}, v]) \in \sigma_i\}
\end{array}$$

Figure 2.8: An implementation of the Add-Wins Set in a PO-Log framework with causal redundancy PO-Log compaction. $o[0]$ denotes the operation's type and $o[1]$ denotes its first argument [Baquero et al., 2017, Figure 8]

2.6.2 Causal Stabilisation and Tagged Causal Stable Broadcast

A timestamp t, and its corresponding message, is said to be causally stable at node i when there can be no more concurrent messages, that is, when all messages subsequently delivered at i will have a timestamp $t' > t$ [Baquero et al., 2017, p. 9]. Since some data types have semantics that rely on concurrent messages, for example the Add-Wins Set, some garbage collection can be made once a message has been identified as causally stable. This is the basis of the second PO-Log compaction technique proposed by Baquero et al. [2017].

To expose causal stability information to the application layer the TCB middleware described in § 2.5.1 is extended to include yet another primitive, *TCS-stable(t)*, which is invoked once a timestamp t is found to be causally stable. This middleware is called the *Tagged Causal Stable Broadcast (TCSB)*. This can be implemented by the middleware by checking, at node i, if a timestamp $t' > t$ has already been delivered at i from every other node j in the set of nodes I [Baquero et al., 2017, p. 9]. Baquero et al. [2017] suggests a possible implementation by defining the function low_i, which returns the greatest lower bound of a timestamp issued at node j delivered at each other node, according to a map L_i that stores the latest timestamp delivered to node i from node j:

$$\mathsf{low}_i(j) \equiv \mathsf{min}(\{L_i(k)(j) \mid k \in I\}).$$

For instance, $\mathsf{low}_i(j) = 4$ means that, at node i, that all other nodes have delivered the fourth message from node j. Baquero et al. [2017] then goes on to define the criteria of causal stability. A timestamp t is causally stable at node i if:

$$t(\mathsf{src}(t)) \leq \mathsf{low}_i(\mathsf{src}(t))$$

where $\mathsf{src}(t)$ denotes the issuer of the message with timestamp t.

With the causal stability information from the TCSB middleware we extend the CRDT scheme from Fig. 2.7 one last time to include the data type-specific function stabilise, as seen in Fig. 2.9. This data type-specific stabilisation function allows us to remove two types of data: stable timestamps and stable operations.

$$\begin{array}{rcl}
\Sigma = T \hookrightarrow O & & \sigma_i^0 = \{\} \\
\mathsf{prepare}_i(o, \sigma_i) & = & o \\
\mathsf{effect}_i(o, t, \sigma_i) & = & \{(t,o) | (t,o) \not\mathsf{R} s\} \cup \{x \in s | x \not\mathsf{R}_0 (t,o) \wedge \\
 & & (t,o) \mathsf{R} s \vee x \not\mathsf{R}_1 (t,o) \wedge (t,o) \not\mathsf{R} s\} \\
\mathsf{R}, \mathsf{R}_0, \mathsf{R}_1 & = & \text{[datatype-specific redundancy relations]} \\
\mathsf{stabilise}_i(t, \sigma_i) & = & \text{[datatype-specific stabilisation function]} \\
\mathsf{eval}_i(q, \sigma_i) & = & \text{[datatype-specific query function]}
\end{array}$$

Figure 2.9: Reference design of a pure operation-based CRDT design based on a PO-Log with redundancy relations R, R_0 and R_1 and stabilise function for PO-Log compaction [Baquero et al., 2017, Figure 10]

The idea behind cleaning stable timestamps, since no more concurrent operations can be delivered after a message has been deemed causally stable, is that the timestamp data is no longer of use; it is always the case for a causally stable timestamp t that later delivered timestamps $t' > t$ [Baquero et al., 2017, p. 14]. This allows, for a timestamp t to be replaced by a least value, denoted as $\bot$—in the case of a vector clock, it would be a timestamp with all zeros, which in practice can be represented by a null pointer value to save memory [Baquero et al., 2017, p. 14].

Baquero et al. [2017] notes that in some data types, like the remove-wins set, there can exist operations which are not redundant when delivered, but becomes redundant once they are causally stable. Therefore, they cannot be removed by the redundancy relations described in § 2.6.1, but can instead be removed in the stabilise function. However, in the Add-Wins Set—the data type of focus in this thesis—no such operations exist. Hence, this technique will not be discussed further in this thesis.

Finally, Baquero et al. [2017] provides an algorithm for combining their framework with the TCS-broadcast. This algorithm can be seen in Fig. 2.10 and allows the CRDT framework to function in, for example, a full-mesh peer-to-peer network, *i.e.* a network where every peer has a direct connection to every other peer. However, as described in § 2.4, it is important that underlying broadcasting middleware is reliable, or else it will not provide SEC as described in § 2.1.

```
on operation_i(o) :
    $m = \text{prepare}_i(o, \sigma_i)$
    $\text{TC-broadcast}_i(m, t)$
on $\text{TC-deliver}_i(m, t)$ :
    $\sigma_i = \text{effect}_i(m, t, \sigma_i)$
on $\text{TC-stable}_i(t)$ :
    $\sigma_i = \text{stabilise}_i(t, \sigma_i)$
```

Figure 2.10: An algorithm of how to combine the TCS-Broadcast middleware together with the CRDT framework [Baquero et al., 2017, Algorithm 2].

2.6.3 Eager Garbage Collection

A deficiency of the causal stabilisation PO-Log compaction technique described in § 2.6.2, is that each node has to wait on messages from other nodes to determine causal stability for a message. Bauwens and Gonzalez Boix [2019] proposes a method to eagerly collect and share causal information within the network. Causal stability for an operation issued by node i, can be determined more eagerly if the nodes that receives an operation send back an acknowledgement to the source of the operation once it has been TC-delivered. When an acknowledgement has been received from all other nodes, causal stability is guaranteed. Node i will then broadcast the causal stability state to all other replicas, which in turn can then remove the relevant meta-data. Bauwens and Gonzalez Boix [2019] notes that while their method allows causal information to be cleaned quicker, it imposes a network overhead due to the extra acknowledgements and sharing of causal information.

The description by Bauwens and Gonzalez Boix [2019] of the eager collection of causal information is unfortunately not more detailed than this. For this reason, a formalisation of an eager causal stability determination algorithm, based on the idea of Bauwens and Gonzalez Boix [2019], is described in § 4.3.

2.7 Peer-to-Peer Connections on the Web

In § 2.4 we introduced the BB and CB broadcasting middlewares, which is later used in § 2.5 to design CRDTs. Both BB and CB assumes a $send$ function and a $receive$ hook, which it can use to send and receive data from other peers. However, as mentioned in Chapter 1, the API surface area of the Web [Mozilla Developer Network, f], called Web APIs, have historically been limited, *e.g.* there exists no TCP socket alternative, where a browser can listen on a port or connect to a peer simply via providing an IP address. Fortunately, a higher level API have recently been standardised—called WebRTC—that allows for direct connections between peers on the Web [Mozilla Developer Network, c]. Thus, instead of using *e.g.* a TCP connection for the underlying broadcast middlewares, we use WebRTC and an interface called `RTCDataChannel`, which is a communication channel with similar semantics to that of TCP, which can be used to transmit arbitrary data [WebRTC Specification].

Connecting two peers however, especially end users who brows the Web, is a complex task, due to firewalls, the lack of public IP addresses or router's who simply do not allow direct incoming connections [Mozilla Developer Network, e]. Despite the aforementioned complexities, WebRTC can achieve direct connections with the help of four key protocols: Interactive Connectivity Establishment (ICE); Network Address Translation (NAT); Session Traversal Utilities for NAT (STUN); and lastly, Traversal using Relays around NAT (TURN) [Mozilla Developer Network, e]. ICE provides the Web browser with a framework to establish a connection with another peer, despite the complexities of real world networks as described above. NAT is used to provide a peer with a public IP address. The router achieves this by keeping a translation table from the peer's private IP address to a publicly accessible one. STUN allows a Web browser to discover the local peer's public IP address, as well as determine restrictions on the peer's router that would hinder a direct connection with a remote peer. This is achieved by the browser sending a request to a specified STUN server which reply with the specified information. At last, a TURN server is used to relay data if no direct connection can be made, and TURN is the protocol over which this takes place.

Important to note is that a peer itself has no use of the information gathered from a STUN server, but rather, the information is needed by another peer to actually initiate a connection. This places the client in a peculiar position, since it needs to send its own STUN information to the other peer, and vice versa, before a connection can be established between them. To solve this, an external, publicly accessible server, is used to channel this data between the two peers [Mozilla Developer Network, d]. This process is called signaling, and the server used to relay the data is called the signaling server.

Chapter 3

Methodology

As Baquero et al. [2017] found, meta-data is often duplicated in the CB-middleware and the application layer, making CRDTs not only wasteful of memory resources but also error-prone. This is due to the fact that causality information has to be embedded manually in every implemented data type, even though different data-types often need the same information. Thus, Baquero et al. [2017] proposes a framework based on a pure op-based CRDT which abstracts the general parts of many CRDTs into a partially ordered log, PO-Log, leaving the developer able to focus on implementing data-type specific semantics. Further on, Baquero et al. [2017] proposes two different garbage collection techniques that aims to both reduce processing and memory resource usage.

Bauwens and Gonzalez Boix [2019] builds upon the work of Baquero et al. [2017] to speed up the garbage collection process by eagerly collecting and sharing causal stability information needed, as explained in § 2.6.2, to perform the causal stabilisation garbage collection technique. Furthermore, Bauwens and Gonzalez Boix [2019] tailor this eager garbage collection to both work in a dynamic full-mesh peer-to-peer network and to be tolerant of transient failures. Dynamic means that nodes can join at any time and transient failures means that nodes might be temporarily be unreachable, but are guaranteed to eventually reconnect. Bauwens and Gonzalez Boix [2019] implement and test their technique in the LuAT framework, a Lua library for distributed programming.

As previously mentioned in § 1.1, the goal of this thesis is to reproduce the findings of Bauwens and Gonzalez Boix [2019], not by re-implementing the eager meta-data removal in Lua, but by designing and re-engineering the same method for meta-data removal on the Web. To achieve this goal, we had to

- **implement** the general CRDT framework by Baquero et al. [2017], in a peer-to-peer setting on the Web, *i.e.* running on a Web browser, thereafter we
- **implement** the garbage collection method proposed by Bauwens and Gonzalez Boix [2019], and at last we
- **evaluated** our implementation to see if (1) the general CRDT framework can at all be adapted to run on the Web over a peer-to-peer network and (2) if similar improvements to memory usage can be seen in the Web environment while applying the garbage collection technique proposed by Bauwens and Gonzalez Boix [2019].

Note that the dynamic join-model will not be implemented nor reproduced.

3.1 Reproduction in Computer Science

Before successful experiments can be considered to be part of science, and interpreted with any confidence, it has to be reproduced and cross-validated at other times and under other conditions [Gómez et al., 2010]. Unfortunately, concerns have recently been raised about the ability to reproduce work within the field of computer science [Gómez et al., 2010; Ivie and Thain, 2018; Vitek and Kalibera, 2011], citing conferences obsession with novelty, the sheer difficulty of actually performing an experiment, as well as proprietary benchmarks and data sets as some possible causes [Vitek and Kalibera, 2011]. Consequently, Vitek and Kalibera [2011] proposes that encouraging more reproduction studies would lead to higher quality research. Furthermore, this notion is supported by Ivie and Thain [2018], that lists six different reasons for why science in computing should be reproducible.

However, there seem to be no consensus of what reproduction entails in computer science and related disciplines [Ivie and Thain, 2018], nor how previous findings ought to be verified [Gómez et al., 2010]. To this end, Gómez et al. [2010] has surveyed other fields to find how different types of reproduction are classified, and found them to fall into three different groups:

1. Those that vary little or not at all with respect to the experiment being reproduced
2. Those that do vary, but that still follow the same method as the experiment being reproduced
3. Those that use different methods to verify the reproduced experiment's result.

The undertakings described in this thesis is thus a reproduction that corresponds to the third category as defined by Gómez et al. [2010], where the aim is to verify the results of the experiments performed by Bauwens and Gonzalez Boix [2019] by using a different method. This has to be the case for two reasons: first, the article by Bauwens and Gonzalez Boix [2019] is not sufficiently described to follow the same method, nor is the source code available due to it being proprietary, and secondly, the LuAT environment is not directly translatable to the Web environment, therefore, deviations must take place. This work will however still be considered an important contribution, not only since it will validate the results found by Bauwens and Gonzalez Boix [2019], but also since it will show whether their technique is applicable to more than one environment, yielding a novel contribution to the field of distributed computing.

3.2 Differences Between LuAT and the Web

An obstacle to reproducing the experiments performed by Bauwens and Gonzalez Boix [2019] is that Lua, and the framework they used, LuAT, cannot run on a Web browser. The Web browser is instead restricted to running JavaScript [Mozilla Developer Network, a] and more lately WebAssembly [Mozilla Developer Network, b], both of which are restricted to Web APIs: the API area available to Web browsers [Mozilla Developer Network, f]. Lua, on the other hand, is an embeddable language where the API area can be extended to any arbitrary system API, depending on the runtime [Lua]. Consequently, a direct replication of Bauwens and

Gonzalez Boix [2019] is difficult, since LuAT may make use of APIs which either have no direct equivalent, or no equivalent at all, in the Web APIs available. An example of this discrepancy is simple peer-to-peer connections. The only Web API available to facilitate peer-to-peer connections is WebRTC [Mozilla Developer Network, c], which is capable, but comes with its own set of limitations; in Lua, on the other hand, a simple TCP socket can be used.

Since no direct replication of the experiments is possible, the re-implementation of the garbage collection mechanism and experiments has to rely on the descriptions of algorithms rather than any concrete implementation. This, unfortunately, sacrifices some validity of this work. Therefore, any deviations from the original implementation in LuAT will be thoroughly discussed and analysed.

3.3 Evaluation Criteria

To determine whether the reproduction of the experiments performed by Bauwens and Gonzalez Boix [2019] was successful or not, we will have to define what reproduction means in this scenario. As stated in § 3.2, a direct reproduction of Bauwens and Gonzalez Boix [2019] is unattainable, and thus, a translation has to be performed. Hence, a reproduction in this scenario is defined as re-creating the algorithms, both the pure CRDT framework by Baquero et al. [2017] and the eager garbage collection by Bauwens and Gonzalez Boix [2019], as explained in the respective papers, mapping concepts to relevant Web APIs as needed. Once the algorithms has been re-created, the experiments that test the effectiveness of the eager garbage collection, performed by Bauwens and Gonzalez Boix [2019], will be re-created in the same manner.

This thesis works after two hypotheses: firstly, that the pure CRDT framework by Baquero et al. [2017] and the eager garbage collection by Bauwens and Gonzalez Boix [2019] can be adjusted to work in the Web environment, and secondly, that the techniques proposed by Bauwens and Gonzalez Boix [2019] is equally effective in this new environment. In this context, equally effective means that the garbage collection can be performed under the same circumstances and that the same relative memory usage and relative memory growth can be achieved, compared to not performing any garbage collection at all.

The relative memory savings are the interesting metric for two reasons: the two different languages, Lua and JavaScript, have different semantics, objects may take a different amount of memory in each, which means an absolute memory usage comparison is of no use. Secondly, different implementations of the same algorithm—which is inevitable due to the reasons mentioned previously—may result in differing absolute memory usage, while the memory complexity ought to be the same.

Success criteria for the two hypotheses are easy to establish: it is successful if the implementation is correct and the results from the experiments correspond to those of Bauwens and Gonzalez Boix [2019]. Correct in this scenario implies that the implemented CRDT provides SEC. It is, unfortunately, more difficult to determine a negative result: since the algorithm is re-created and not directly replicated, it may either be a fault in the implementation or an inherent flaw in the Web environment, which leads to a discrepancy in test results, with no simple way of knowing which one it is. In this case, further investigation will be required with additional metrics: can meta-data be garbage collected at all? If yes, then why is there a discrepancy between the results from this thesis and those of Bauwens and Gonzalez Boix [2019]. If meta-data cannot be garbage collected at all, then there is most likely a fault in the implementation, rather than the environment.

Chapter 4

Design and Implementation

The eager garbage collection and subsequent experiments, performed by Bauwens and Gonzalez Boix [2019], was implemented in a programming environment called LuAT—a Lua library for distributed programming. LuAT allows actors to communicate over a "mobile ad hoc network" in a distributed setting, using an actor-based programming model. Further, Bauwens and Gonzalez Boix [2019] mentions that their implementation of network discovery and CRDT replication piggybacks on top of the network publication and discovery mechanism provided by LuAT, and they use this to set up their full-mesh peer-to-peer network using a dynamic join-model, *i.e.* that peers can join the network at any time.

Unfortunately, no further description is made of how LuAT provides these mechanisms, nor of the underlying protocols used. This makes it impossible to accurately reproduce a similar network that can be used to perform the experiments. However, Bauwens' and Gonzalez Boix's [2019] implementation is all built on-top of a reliable causal broadcast-middleware and the purpose of a middleware is to provide a convenient programming model that masks heterogeneity [Coulouris et al., 2012, p. 17]. This means that by implementing a reliable causal broadcasting middleware, any eventual differences at the network layer between Bauwens' and Gonzales Boix's [2019] implementation and another implementation would be masked and the algorithm should thus work regardless of the underlying network. This implies that, while efficiency might differ due to disparities at the network layer, correctness should not.

4.1 System Model

Before a description of concrete design and implementation details are described, we will formalise the system model used to implement the network and subsequent algorithms, in order to lay out what guarantees can be made of the system as a whole. Since this work described in this thesis aimed to implement and reproduce the algorithms described by Bauwens and Gonzalez Boix [2019], we have opted to adopt their system model, which in turn is inspired by the system model used by Baquero et al. [2017] explained in § 2.2. As this work did not aim to reproduce the dynamic join model described by Bauwens and Gonzalez Boix [2019], simplifications has been made to the system model where it makes sense.

Our system model assumes a full-mesh peer-to-peer network consisting of N nodes.

Each node in the system is a separate machine, physical or virtual, that communicates with other nodes over a reliable data channel, *i.e.* messages are guaranteed to be transmitted without duplication or corruption. Different from Bauwens and Gonzalez Boix [2019], we assume a static network that will not change once established with N peers. After the network has been set up, each node instantiates and hosts a replica of a shared Add-Wins Set—it is assumed that all replicas are in an equivalent and consistent initial state, before modifications are made to the set. We assume that any failures are temporary and expect that a faulty node will recover eventually. Messages that cannot be sent during the temporary failure are buffered and will be correctly read by the receiving node once the faulty node has recovered.

The assumed failure model has important implications as it causes the B-broadcast and subsequently C-broadcast to be reliable, as explained § 2.4. If the broadcasts were not reliable, then the Add-Wins Set would break SEC, which means that replicas could diverge and possibly never converge again.

4.2 Setting Up a Peer-To-Peer Network

The network layer for this work is built on-top of WebRTC, a set of standards which make it possible for Web browsers to send data peer-to-peer without resorting to third-party software [Mozilla Developer Network, c], and is further explained in § 2.7. In that section, it is also mentioned that WebRTC provides no pre-defined means of negotiating connections, and thus, a signaling server was designed to provide both peer discovery and signaling.

The WebRTC specification does not specify how signaling should be done, but suggests two different, potential means of communication: `XMLHttpRequest` (XHR) and Web Sockets [WebRTC Specification]. The signaling server designed for this work utilises Web Sockets to communicate with potential peers, as it, in contrast to XHR, provides bidirectional communication [WebSocket Protocol]. The benefit of a bidirectional communication channel like Web Sockets is that the signaling server can notify a client when needed, *e.g.* when a new peer connects, whereas XHR would require all clients to continuously poll the server to see if new data has arrived—a process that would both waste bandwidth and potentially be slower.

The signaling server can be implemented in any programming language and runtime, as long as it supports Web Sockets. Ultimately, the signaling server was implemented in JavaScript and runs on the runtime Node.js[1]. The reasoning behind this is that, if the signaling server and the eager CRDT implementation is written in the same language, the same semantics and assumptions can be made between the two parts. This would make the system, as a whole, easier to reason about. While other programming languages may be better suited to this task in other ways, *e.g.* performance, it was preferable to keep complexity low, since it will not be used after the network has been set up.

The peer discovery and signaling process is a six step process:

1. A client connects to the signaling server via a WebSocket (WS) connection. The server assigns this client a globally unique id i.
2. All other clients $j \neq i$, if any, are notified that a new client i has been discovered.

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

3. The client j start the process of initiating a WebRTC connection by creating a WebRTC *offer* with meta-data about what types of connections this peer support. This is sent to client i via the signaling server. The signaling server do not process the message, only relays it to client i.
4. Client i will create a WebRTC *answer* to client j's offer. This will be sent to client j via the signaling server.
5. Both client i and client j will start to generate WebRTC *ice candidates*, describing potential methods to initiate a connection to the other peer as explained in § 2.7. The ice candidates are sent to the other peer, once again, via the signaling server.
6. The clients will apply ice candidates from the other peer as they trickle in and try to establish a connection continuously.

This process is also illustrated in Fig. 4.1. Once the clients are connected, all communication is done through a `RTCDataChannel` object to transmit arbitrary data, which is a communication channel that is reliable by default, providing similar semantics to that of TCP [WebRTC Specification]. This is important because the broadcasting middlewares described in § 2.4 assume that the underlying communication channel to be reliable.

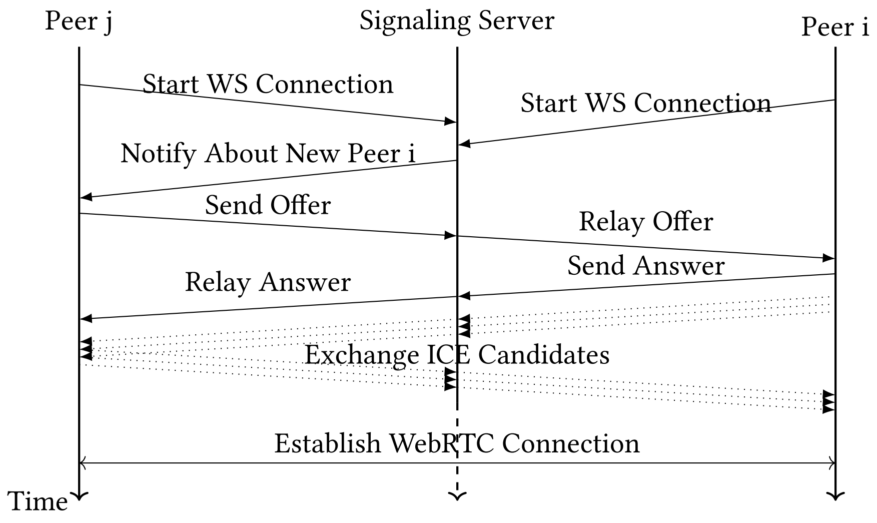


Figure 4.1: Illustration of how two peers initiate a WebRTC connection with the help of a signaling server.

4.3 Formalising Eager Garbage Collection

The eager garbage collection technique proposed by Bauwens and Gonzalez Boix [2019] is unfortunately not formally described nor proven to be correct. Only the general idea is described: if the receiver of an operation sends back an acknowledgement once the operation has been processed, the sender of the operation can establish the operation to be causally stable once it has received an acknowledgement from all other peers. Once this happens, the sender should share the causal stability information with all other peers so that they too can remove any meta-data associated with the newly stable operation.

There are three main pieces of information lacking from their description that hinders an implementation, namely:

1. Must the acknowledgement messages be delivered in a specific order? Both regarding other acknowledgement message and operation messages.
2. Are there any specific ordering requirement for the sharing of causal stability?
3. Since causal stability is established at a specific node—the sender node in this case—can the system guarantee that the timestamp will also be causally stable at the receiving node once it receives such a message?

Due to the lacking description of the eager causal stability determination by Bauwens and Gonzalez Boix [2019], a formal description of an algorithm inspired by the idea of eagerly collecting and sharing causal stability, together with a proof of its correctness, is described here.

4.3.1 Eager Causal Stability Detection Algorithm

The eager garbage collection technique is about eagerly finding causal stability so that the causal stabilisation PO-Log compaction described in § 2.6.2 can be applied. First, we define the Eager Tagged Causal Stable Broadcast (ETCSB), which satisfy the TCSB interface with the following primitives: ETCS-broadcast, ETCS-deliver and ETCS-stable. The ETCSB build upon the TCB defined in § 2.5.1 and can be implemented with the following algorithm:

```
on initialise
    let ACKCount $\in T \hookrightarrow \mathbb{N}$

on ETCS-broadcast($m$) at node $i$:
    $t_i$ = TC-broadcast(<MSG, $m$>)
    ACKCount[$t_i$] = 1

on TC-deliver(<$t_i$, <MSG, $m$> >) at node $j \neq i$ from node $i$:
    ETCS-deliver(<$t_i$, $m$>)
    $send^i_j$(<ACK, $t_j = V_j$, $t_i$>)

on $receive^i_j$(<ACK, $t_j$, $t_i$>) at node $i \neq j$ from node $j$:
    wait until $t_i \leq V_i$
    ACKCount[$t_i$] += 1
    wait until ACKCount[$t_i$] == $|I|$
    delete ACKCount[$t_i$]
    ETCS-stable($t_i$)
    TC-broadcast(<STABLE, $t_i$>)

on TC-deliver(<_, <STABLE, $t_i$> >) at node $j \neq i$:
    ETCS-stable($t_i$)
```

When initialising the ETCSB we create a map ACKCount from timestamps T to the natural numbers $\mathbb{N}$. The idea is that this map will be used to keep track of how many nodes has acknowledged a certain message, which in turn is the information needed to decide causal stability.

When a node i broadcasts a message through the ETCS-broadcast primitive, we let the underlying TC-broadcast handle it. We also note the timestamp t_i attached to the message, and store this in the map ACKCount with an initial value of 1 to denote that one node has processed the message—that is, the sending node i. Since there now exists more message types, we append the message with the header MSG to denote that it is a normal message which should be delivered to upper layers in the middleware stack.

Once the message has been TC-delivered at node $j \neq i$, we can ETCS-deliver it as well. At the same time, we notify the original sender i that the message with timestamp t_i has indeed been processed. Important here is that we attach the current timestamp $t_j = V_j$ at node j to the message; since this is a one-to-one send operation we cannot rely on the underlying TCB to causally order the message but rather have to implement it directly by attaching the current timestamp. Note that we do not increase the clock value as that would cause other nodes in the network to wait for the ACK-message that they will never receive, due to it being a one-to-one send operation, and not broadcast to all.

When the original sender i receives the acknowledgement message from node j, it will wait until all causally preceding messages has been delivered and processed by comparing the attached vector clock; it is important to respect the causal ordering since concurrent messages can still exist in the hold-back queue in the underlying TCB. Once this criteria has been fulfilled, we increase the counter in ACKCount with key t_i by one. This procedure is repeated until we have received an acknowledgement from all other nodes in the set of nodes I, which is fulfilled once the counter has the value $|I|$. Since we have received an acknowledgement from all other nodes, we can conclude that the operation with timestamp t_i is causally stable. Thus, we can invoke the primitive ETCS-stable to notify upper layers in the middleware stack, as well as broadcasting, over the TCB with the STABLE header, that the operation with timestamp t_i now has become stable.

Once the TCB delivers a message with the STABLE header at another node $j \neq i$, we can conclude that the message with timestamp t_i is also causally stable at node j. This is thanks to the underlying TCSB delaying delivery of the STABLE message until all messages concurrent to m_i has been delivered—further explanation of this is found in the proof in § 4.3.2.

An example run of the algorithm is illustrated in Fig. 4.2 between many nodes, where the algorithm tries to decide the stability of message m_i at all nodes. In this example, message m_j and m_k are both concurrent to m_i, but not concurrent to each other as $m_j \rightarrow m_k$, which means that we can see how the algorithm delays the delivery of three messages: message m_k; the ACK message ACK(m_i) from node k; and finally the STABLE message STABLE(m_i) at node j. Note that at all nodes, any concurrent message to m_i has been delivered before establishing causal stability.

4.3.2 Proof of Correctness

To prove the correctness of this algorithm we need to show that, (1) message m_i is stable at node i before the algorithm invokes the ETCS-stable hook at node i, and (2) the message m_i is stable at all nodes $j, k \dots l \neq i$ before the algorithm invokes the ETCS-stable hook at nodes $j, k \dots l$. To prove both, we rely on the definition of causal ordering and vector clocks as described in § 2.3.

To prove (1), we need to prove that at least one ACK message to m_i is processed after the last message concurrent to m_i has been delivered. In the algorithm, the first 'wait' in 'on receive(<ACK, t_j, t_i>)' make sures the any ACK is processed only when node i's vector clock causally succeeds it, according to the value of their timestamp. This reduces our proof to proving that any message $m_{j,k,\dots l}$ concurrent to m_i has a

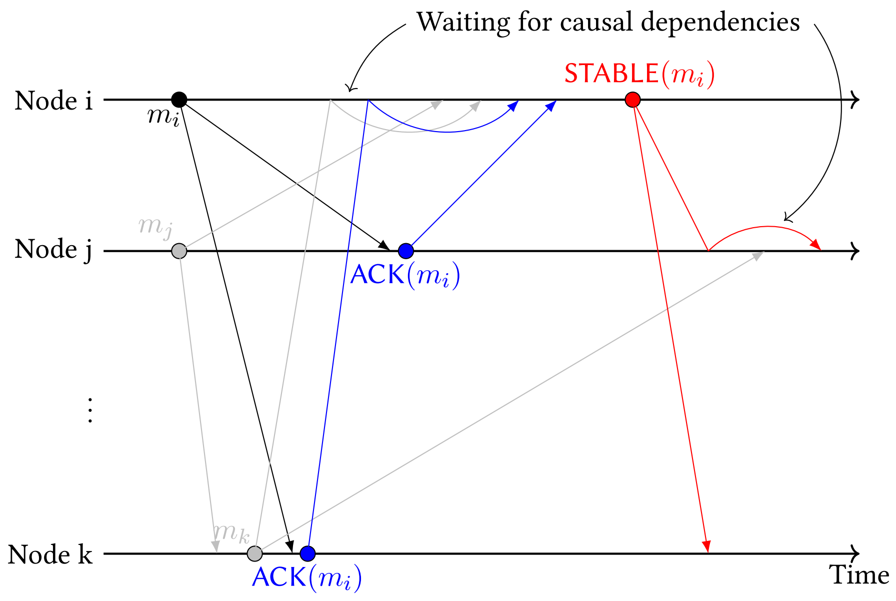


Figure 4.2: Illustration of the eager causal stability determination algorithm between many nodes—of which only nodes i, j and k are depicted—for the message m_i. The messages give rise to ACK messages and consequently STABLE messages. The messages m_j and m_k are concurrent to m_i and will in turn give rise to ACK messages, but these are omitted in this illustration for clarity.

timestamp strictly less than the timestamp of at least one ACK message, and can be formulated as

$$T(m_j), T(m_k) \dots T(m_l) < T(\mathsf{ACK}(m_i)) \tag{4.1}$$

where $T(m)$ is the timestamp of a message m and ACK(m) is the acknowledgement of a message m. To prove (2), remember that the TCB will deliver all messages in a causal ordering as described by their vector clock timestamps, which reduces this proof to proving that any messages $m_{1 \dots N}$ concurrent to the stable message m_i has a timestamp that strictly precedes the timestamp of the STABLE message of m_i. This can be formulated as

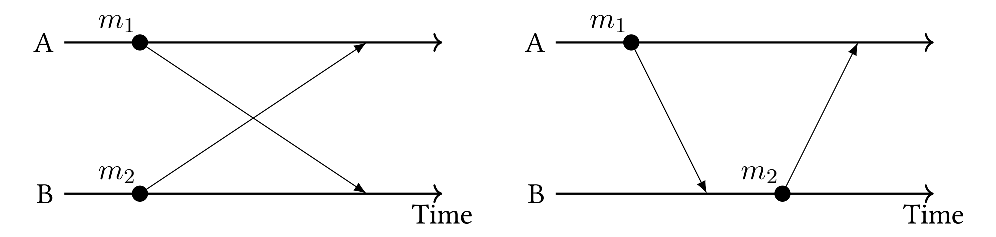
$$T(m_1), T(m_2) \dots T(m_N) < T(\mathsf{STABLE}(m_i)) \tag{4.2}$$

where STABLE(m) is the STABLE message of the message m.

Our first observation is that any concurrent message $m_{j,k \dots l}$ to m_i must have the causal relation

$$\begin{aligned} send^i_j(m_j), send^i_k(m_k), \dots, send^i_l(m_l) \rightarrow receive^{j,k \dots l}_i(m_i) \Leftrightarrow \\ send^i_{j,k \dots l}(m_{j,k \dots l}) \rightarrow receive^{j,k \dots l}_i(m_i) \end{aligned} \tag{4.3}$$

since otherwise the messages would causally succeed the message m_i instead of being concurrent. An example of this is illustrated in Fig. 4.3, where Fig. 4.3a shows two concurrent operations and Fig. 4.3b shows two non-concurrent messages. In the figure, we can clearly see that a node must receive a concurrent message after it sends its own message, as it otherwise would not be concurrent at all.



(a) For operations $m_1 \parallel m_2$ it is always the case that $send_A^B(m_1) \rightarrow receive_B^A(m_2)$ and $send_B^A(m_2) \rightarrow receive_A^B(m_1)$.

(b) For any two operations $m_1 \rightarrow m_2$ it is always the case that $receive_A^B(m_1) \rightarrow send_B^A(m_2)$.

Figure 4.3: Illustration between the differences in the send and receive order between concurrent and non-concurrent operations.

Our second observation is that the nodes $j, k \dots l$ will receive the message m_i before sending a corresponding ACK message, *i.e.*

$$receive_i^{j,k\dots l}(m_i) \rightarrow send_{j,k\dots l}^i(\mathsf{ACK}(m_i))$$

which finally, thanks to the $\rightarrow$ relation's property of transitivity, can be combined with Eq. (4.3), which yields

$$send_{j,k\dots l}^i(m_{j,k\dots l}) \rightarrow receive_i^{j,k\dots l}(m_i) \rightarrow send_{j,k\dots l}^i(\mathsf{ACK}(m_i)). \tag{4.4}$$

Remember that according to the algorithm, each ACK message sends the current timestamp—without altering it—along with the message. This means that

$$T(m_{j,k\dots l}) < T(\mathsf{ACK}(m_i)) \tag{4.5}$$

due to the fact that receiving the message m_i bumps the vector clock, and as can be seen in Eq. (4.4), this happens before sending and timestamping the ACK message. This is the same as Eq. (4.1), which is what we needed to prove (1).

To also prove (2), we note that once the last ACK message has been received, the algorithm will broadcast, over the TCB, a STABLE message, which means that

$$receive_{j,k\dots l}^i(\mathsf{ACK}(m_i)) \rightarrow send_i^{j,k\dots l}(\mathsf{STABLE}(m_i)) \Leftrightarrow$$
$$T(\mathsf{ACK}(m_i)) < T(\mathsf{STABLE}(m_i)).$$

This fact can be combined with Eq. (4.5), which results in

$$T(m_{j,k\dots l}) < T(\mathsf{ACK}(m_i)) < T(\mathsf{STABLE}(m_i)) \Rightarrow$$
$$T(m_{j,k\dots l}) < T(\mathsf{STABLE}(m_i)) \Leftrightarrow$$
$$T(m_j), T(m_k) \dots T(m_l) < T(\mathsf{STABLE}(m_i))$$

which is the same in Eq. (4.2), which is what we needed in order to prove (2).

Chapter 5

Evaluation

To evaluate the hypotheses stated in § 3.3, we need a way to set up experiments that involve many nodes, coordinate between them and measure their memory usage accurately across different garbage collection implementations. In § 5.1, we describe the challenges and solutions of how to measure memory on the Web; in § 5.2 we describe the test bench used that automatically set-ups and controls the experiments; in § 5.3 we describe the different garbage collections techniques used and the differences between them; and finally, in § 5.4 we describe the necessary steps we had to take to produce comparable results.

5.1 Measuring Memory on the Web

Memory usage on the Web can be defined in many ways, and three of these was considered during the evaluation:

1. The number of bytes allocated on the underlying JavaScript Virtual Machine's (VM's) heap.
2. The number of bytes allocated by the Web browser process.
3. The number of bytes of the object graph of the Add-Wins Set.

All these different measures of memory are relevant in different, subtle ways. For example, Item 1 accurately represents the actual memory in use by the JavaScript VM, and is what Bauwens and Gonzalez Boix [2019] measured in their experiments. This makes it, in terms of reproducing their work, the most relevant method of measuring memory. However, in a memory constrained environment, the most relevant method is Item 2 since that is what directly impacts system memory needs—unfortunately both this measure and Item 1 makes it difficult to draw any conclusions of the effectiveness of different garbage collection implementations as there are many types of data, other than the Add-Wins Set, allocated on the heap that are included in the measurement. Instead, Item 3, may be appropriate, since it allows fine-grained control over what is included in the measurement, which makes the comparison between different garbage collection implementations more fair. Even so, this measure hides total memory usage—*e.g.* one implementation might use half as much memory as another implementation, but if that memory only accounts for a small share of total memory usage, any improvements to the algorithm are insignificant, as the total system memory needs barely changes. Since Item 1 is the measurement used by Bauwens and Gonzalez Boix [2019] and we strive to as accurately as possible reproduce their work, this was partly used to measure memory usage. To tackle the deficiency of Item 1—that other objects which we have no control over will also be measured—the measurements are complimented with Item 3.

Measuring memory usage of the JavaScript VM, is unfortunately difficult, due to the limited APIs available, as previously discussed in § 3.2. Namely, there exist no documented way to measure the size of objects allocated on the JavaScript heap, nor the size of an object graph. Though, fortunately for this work, is that Chromium[1], an open source Web browser, has an undocumented command line switch `enable-precise-memory-info`[2] which was used to expose the total size of objects currently allocated on the heap. Unfortunately, this limited the testing to only one Web browser as no equivalent methods of measuring memory was found for those, though it is a reasonable assumption that other Web browsers behave similarly.

When Bauwens and Gonzalez Boix [2019] measure the Lua heap, they forced the Lua runtime to perform a garbage collection to remove any dead objects from the measurements. To make our reproductions as accurate as possible, we also request the JavaScript VM to perform a garbage collection before each measurement. Just as when measuring the memory on a heap, requesting a garbage collection is not accessible via a JavaScript API, but can be exposed in V8, the JavaScript VM used by Chromium, by passing the command line switch `expose-gc`[3].

The command line switch `enable-precise-memory-info`, unfortunately does not allow us to measure the size of an object graph. Instead, we chose to approximate the size of the object graph by serialising the object to another format, whose size can be measured. We opted to serialise the object into an equivalent JSON string representation, using the `JSON.stringify`[4] function and measuring the length of this string. While more compact serialisation representations are possible, *e.g.* serialising into a byte-array, `JSON.stringify` was favoured thanks to its simplicity due to being part of the JavaScript language specification. Furthermore, while the serialised object graph depicts changes in the algorithms' memory usage, and can be used to explain whether memory allocation is because the underlying VM or the algorithm, it is not representative of the system memory requirements and will thus not be used for direct or relative comparisons between our results on the Web versus the LuAT results by

Bauwens and Gonzalez Boix [2019].

To test whether these measurement methods, along with the undocumented command line switches work as expected, a simple experiment was conducted. The experiment made a total of 500 measurements, both the heap size memory and the serialised size memory. After 100 measurements, the experiment allocated an array filled with $250,000$ numbers, which should, assuming each number is stored in 4 bytes, allocate one megabyte of memory. After yet another 100 measurements, we remove the allocated array of memory, and continue to measure 100 times to see if the memory is indeed gone. We then start to allocate $250,000$ numbers gradually over 100 measurements, which should result in a linear increase of memory over

[1]`https://www.chromium.org/`

[2]`https://cs.chromium.org/chromium/src/content/public/common/content_switches.cc?q=kEnablePreciseMemoryInfo`

[3]`https://chromium.googlesource.com/v8/v8/+/refs/tags/8.3.84/src/flags/flag-definitions.h#1088`

[4]`https://tc39.es/ecma262/#sec-json.stringify`

time. At last, we clear any allocation again for the last 100 measurements.

The results of this test can be seen in Fig. 5.1, where indeed can observe the expected behaviour, both for the heap size in (A) and the serialised JSON size in (B). As previously explained, the serialised version is a non-compact representation of the object graph—this is visible in the graph, indeed, an allocation of 250000 increases the serialised size by more than 1.5Mb, assuming each character in the JSON string to be one byte, instead of the real size of 1Mb. Note, however, that while the JSON representation does not accurately depict the total memory allocated, it clearly shows the general behaviour of memory growth.

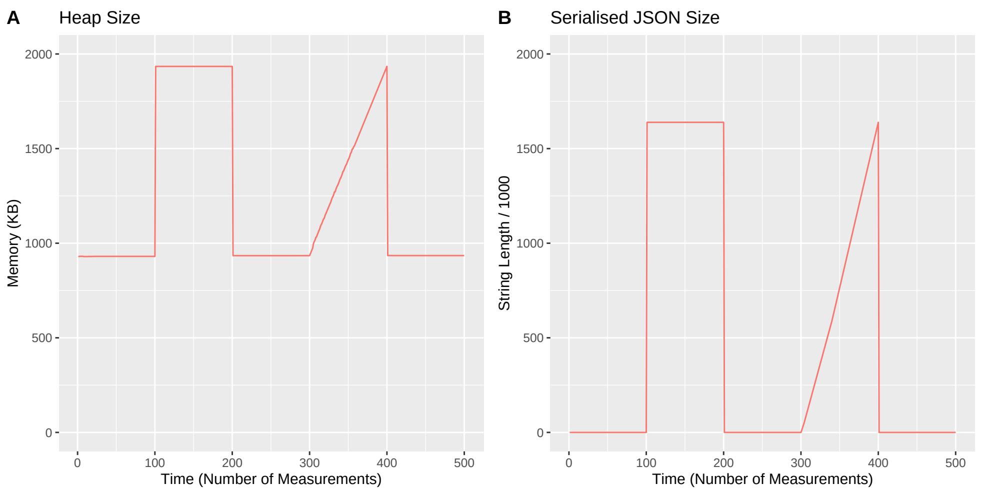


Figure 5.1: Results from the different memory measurement methods. (A) shows the total size of objects allocated on the heap as reported by the VM and (B) shows a direct measurement of the serialised object graph.

5.2 Test Bench

To appropriately measure the memory, we needed a tool that can automatically set up, coordinate, measure, and tear down several nodes in a network. To achieve this, a *test driver* was built that spawns a headless version of the Web browser Chrome[5]—a derivative of Chromium browser mentioned in § 5.1—which is controlled by the Web automation framework Selenium WebDriver[6]. Additionally, the test driver also contains a HTTP server that allows it to be remotely controlled. Finally, the test driver was packaged in a container that allows the test bench to spawn several test driver instances, all of which can be controlled remotely thanks to the built-in HTTP server. Each instance of a test driver is then supposed to act as a node in the peer-to-peer system. Remember that Web browsers need a signaling server to establish a P2P WebRTC connection, as explained in § 4.2, consequently, the signaling server was also built as a container, to enable set-up and tear-down of the system as a whole.

To orchestrate and coordinate between instances of the test drivers and signaling server, a *test controller* was developed. The test controller's job was two-fold:

[5]`https://www.google.com/chrome/`
[6]`https://www.selenium.dev/`

- Spawn an instance of the signaling server and an appropriate number of instances of the test driver for each experiment.
- Coordinate node actions and request nodes to measure their memory usage via the test drivers HTTP server.

The complete test bench system, and its interactions, can be seen in Fig. 5.2.

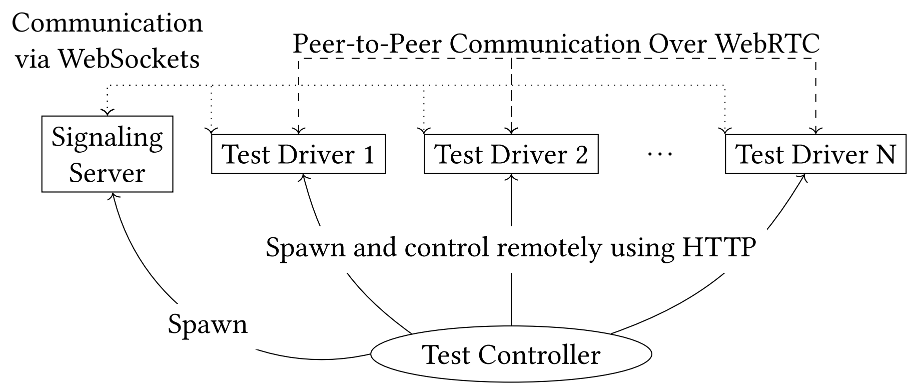

Communication via WebSockets
Peer-to-Peer Communication Over WebRTC
Signaling Server
Test Driver 1
Test Driver 2
· · ·
Test Driver N
Spawn and control remotely using HTTP
Spawn
Test Controller

Figure 5.2: Illustration of how the test controller spawns one instance of a signaling server and several test driver instances, and of how all parts of the system communicate with each other.

All tests were run on a Linux host with 8GB of RAM and an Intel i7-5500U CPU. All experiments used Chrome version 80.

Some experiments test the behaviour of the system during partial failures, which require us to toggle a node's connectivity status to offline/online. The toggling of connectivity status is simulated in the application layer by a boolean flag, which starts to buffer any incoming and outgoing operations, which means that the WebRTC connection between peers is never broken and thus never re-established. The reason for simulating the connectivity status is that it simplified the implementation of both the network layer and the test bench itself, since we do not have to implement re-connection logic and we do not have to break the physical internet connection between containers, respectively. This simplification does not contradict the system model described in § 4.1, but is rather a simplified method to test if the algorithms works correctly in the environment described in the system model.

5.3 Garbage Collection Implementations

Just as Bauwens and Gonzalez Boix [2019] did, we measured memory usage of the Add-Wins Set—built upon an implementation of the pure-op based CRDT framework by Baquero et al. [2017]—between different garbage collection implementations. As mentioned in § 2.6.3, the difference between the garbage collection implementations is how the causal stability for an operation is determined; this determination is performed in the TCS-broadcast middleware. A TCS-broadcast, or TCSB, as described in § 2.6.2, has an interface consisting of a broadcast function, a stable hook, and a delivered hook. The different variants of the TCSB all implement the same interface which means they can be used interchangeably by the CRDT framework without any modifications. To distinguish between the different TCSB implementations, they have been named as the following:

1. *Original Tagged Causal Stable Broadcast* (OTCSB): the implementation as originally suggested by Baquero et al. [2017] and explained in § 2.6.2.
2. *Non-Stabilising Tagged Causal Stable Broadcast* (NTCSB): an implementation that does not keep track of causal stability at all. This means the causal stabilisation PO-Log compaction cannot be performed, and is used to obtain a baseline to compare against. Implementation-wise it is equivalent to the TC-broadcast explained in § 2.5.1 with the difference that it also provides the stable hook to satisfy the interface required by the CRDT framework, despite the hook never being invoked since stability is not tracked.
3. *Eager Tagged Causal Stable Broadcast* (ETCSB): this implements the eager causal stability determination proposed by Bauwens and Gonzalez Boix [2019], and is further described in § 4.3.

5.4 Experiment Preparation

When performing experiments to either prove or disprove our hypotheses, we noticed some inconsistencies in our results, namely, that memory seemed to grow faster in the start of the experiment than towards the end. The most likely reason for those is that the underlying JavaScript VM optimises internal data structures, *e.g.* preallocation memory, resulting in a higher memory growth in the start in the experiment compared to later in the experiment when no more optimisations can be done. This proved to be problematic as it would be difficult to directly compare our results to that of Bauwens and Gonzalez Boix [2019]. To confirm if this was the case, we performed an additional experiment to see if such a "stabilisation phase" could be found, and how this information could be leveraged to achieve comparable results. The results from this experiment, explained in § 5.4.1, is important as it influences the design of our other experiments.

Furthermore, when analysing the descriptions and the results of the experiments performed by Bauwens and Gonzalez Boix [2019], it is not obvious if the causal redundancy PO-Log compaction technique described in § 2.6.1 was performed at all. Whether it was performed should lead to different results—causal redundancy removes redundant operations which means that, *e.g.* in an Add-Wins Set, adding the same element many times should not result in higher memory usage, as redundant operations are discarded. Conversely, memory usage should increase in the same situation if causal redundancy is not applied. With this in mind, and analysing the results of Bauwens and Gonzalez Boix [2019], it seems like the causal redundancy PO-Log compaction technique was not performed. To confirm this suspicion, an experiment was designed and conducted. Similarly to the aforementioned inconsistency, the results from this experiment, presented in § 5.4.2 is important, as it too influences the design when reproducing the experiments by Bauwens and Gonzalez Boix [2019].

5.4.1 Engine Stabilisation Behaviour

As previously mentioned in § 5.4, we noticed some inconsistencies in our initial results between the first measurements of an experiment and later measurements of the same experiment. We had a suspicion that this was due to the engine going through a "stabilisation" phase, which may include pre-allocation space for buffers and containers. This inflated the memory usage in the heap measurements, which made it difficult to directly compare our measurements to the results of Bauwens and Gonzalez Boix [2019].

To confirm this suspicion, we designed an experiment that stressed the system by repeatedly inserting and clearing elements in a shared Add-Wins Set. Each “round” consisted of inserting the numbers 1–1000 into the set, and then sending a clear operation to reset any state—the nodes in the system inserts one number each in an interleaved fashion, *i.e.* the nodes take turns inserting one number at a time. If the system’s memory usage plateaus after a certain amount of rounds, we can conclude that the anomalies observed indeed was due to underlying engine optimisations, and not due to *e.g.* a memory leak. If this indeed is the case, we can perform the same amount of stabilisation rounds before each experiment to gain accurate measurements of memory growth due to the algorithm, instead of the underlying engine pre-allocating memory. This was done for all implementations listed in § 5.3 to investigate whether the stabilisation phase affects them differently.

The results from this experiment is shown in Fig. 5.3. As can be seen in the plot, the memory usage of each implementation grows significantly the first 10 stabilisation rounds, and plateaus after around 15–20 rounds. We also note that the ETCSB implementation has a higher average memory usage after stabilisation—despite the clear operation that should reset all implementations to a roughly equal usage of memory. This is most likely a result of the engine optimising the implementations differently.

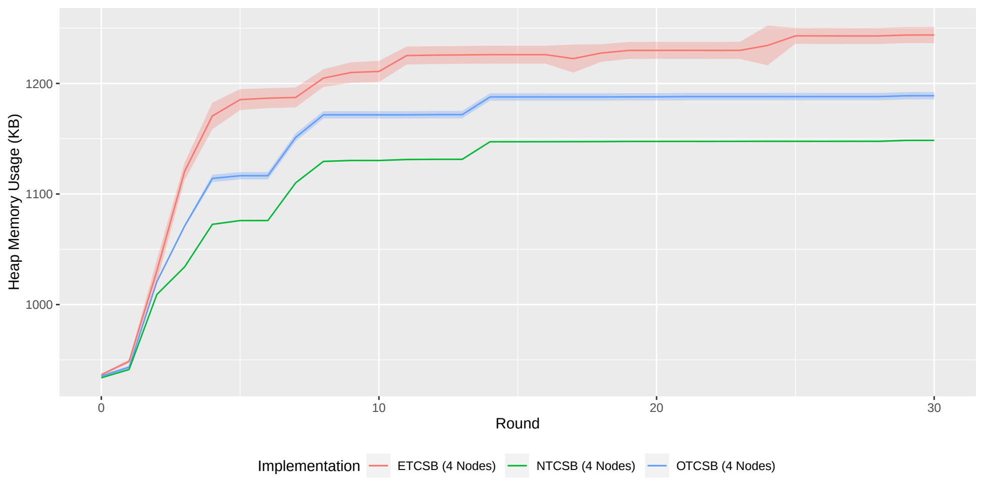


Figure 5.3: Heap memory usage over 30 stabilisation rounds. Each round uses 10 measurements and these are shown as points in the graph. The line illustrates each implementations moving average and the shaded area represents the standard deviation.

The results from this test shows three things: the first is that a stabilisation phase is indeed necessary to get reliable results, we want to measure the differences in memory used by the algorithms, not what the underlying engine allocates; secondly, performing this stabilisation 20 rounds should be enough to get the system into a stable state; and finally, the ETCSB seems to be more affected by this stabilisation phase than the other implementations. From now on, before any experiment that measures heap memory usage, it is assumed that we have performed a stabilisation phase of 20 rounds.

5.4.2 Causal Redundancy Behaviour

In § 5.4 we mentioned that, despite the description of the experiments performed by Bauwens and Gonzalez Boix [2019], no causal redundancy PO-Log compaction seemed to be performed, for any implementation. This would have an immense effect in test results as sending redundant operations should not affect memory usage while non-redundant would. To confirm this suspicion, we designed and performed an experiment that either (1) deliberately sends operations that are causally redundant or (2) send operations are not causally redundant. We then compare the results to the results of Bauwens and Gonzalez Boix [2019] to see which versions are the most similar.

To deliberately send operations that are causally redundant we chose to replicate their first experiment, where they measure memory usage between TCSB implementations, as close as possible. In this experiment, they "performed 1000 add operations on the [add-wins] set, repeatedly selecting numbers between 1 and 100", and where "every 100 operations the source node for the operations was switched". Since numbers between 1 and 100 was repeatedly inserted, we would expect causal redundancy to take place, assuming they are integer numbers. A benefit of replicating their experiment directly is that a baseline is established that can properly compare against the results of Bauwens and Gonzalez Boix [2019]. Furthermore, to gather data where no operations are causally redundant, we performed the same test, but instead of repeatedly inserting a number between 1 and 100, we inserted a linearly increasing range of number from 1 to 1000, so that causal redundancy never could be performed.

This experiment used the NTCSB implementation, as explained in § 5.3, which means that the causal stabilisation PO-Log compaction technique was never performed. We use NTCSB because it is easier to see the effect of causal redundancy since other implementations also perform the causal stability PO-Log compaction, that would interfere with the results.

The average memory usage and standard deviation of running this experiment 20 times can be seen in Fig. 5.4, where the test without causal redundancy continues to grow indefinitely, while the test with causal redundancy stops growing once no more unique elements are inserted after 100 operations. Indeed, our assumption about no causal redundancy being applied in the tests of Bauwens and Gonzalez Boix [2019] seems to be correct, as our result where no causal redundancy was performed correspond to their results. The sudden increase in memory usage at around operation 100 in chart (A), and subsequent smaller sudden increments, are most likely due to the JavaScript VM resizing containers, since they occur roughly at the same insertion across all 20 runs—note the low standard deviation—and do not show up in the serialised object graph measurements shown in graph B.

From this experiment we can conclude that the experiments performed by Bauwens and Gonzalez Boix [2019] did not perform any causal redundancy PO-Log compaction. Indeed, after asking Bauwens and Gonzalez Boix [2019] directly, it was confirmed that causal redundancy was disabled in order to more accurately measure the effects of only causal stabilisation [Bauwens, 2020]. Hence, we will tweak the corresponding experiments so that causal redundancy is not performed. Any such deviations from the original experiment design will be explained and motivated.

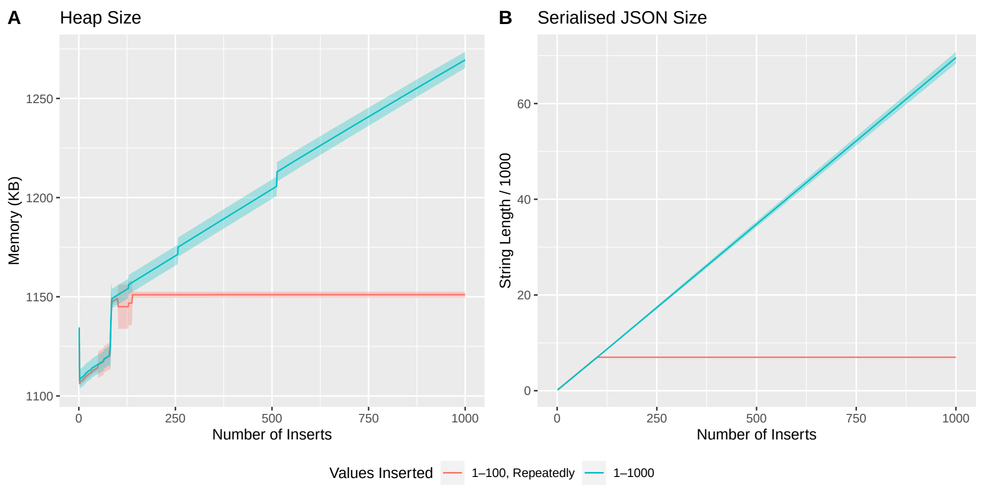


Figure 5.4: Differences between repeatedly inserting the same elements between 1–100, triggering the causal redundancy PO-Log compaction and inserting unique numbers between 1–1000, where no causal redundancy can be performed. The line represents the mean of 20 runs and the shaded area represents the standard deviation.

Chapter 6

Results & Discussion

As stated in § 3.3, this thesis works after two hypotheses: that the pure CRDT framework by Baquero et al. [2017] and the eager garbage collection by Bauwens and Gonzalez Boix [2019] can be adjusted to work in the Web environment, and secondly, that the techniques proposed by Bauwens and Gonzalez Boix [2019] is equally effective in this new environment. Any experiments of our implementation should then either help prove or disprove these hypotheses.

To test our first hypothesis—that the pure CRDT framework and the eager garbage collection by Bauwens and Gonzalez Boix [2019] can be adjusted to work in the Web environment—we ran a stress test to validate the algorithm's correctness. This is described in § 6.1.

To test our second hypothesis—that the eager garbage collection is equally effective in the Web environment as it was in LuAT—we reproduced two experiments by Bauwens and Gonzalez Boix [2019]. The first experiment tested the effectiveness of the ETCSB implementation against the other two TCSB implementations described in § 5.3, and the second tested the behaviour of the ETCSB implementation compared to NTCSB implementation, in an environment with temporary outages. These experiments are described in § 6.2 and § 6.3 respectively.

6.1 Correctness Validation

The purpose of this test was to confirm our first hypothesis: that the pure CRDT framework by Baquero et al. [2017] and the eager garbage collection by Bauwens and Gonzalez Boix [2019] can be adjusted to work in a P2P Web environment. To validate correctness we had to investigate two properties: (1) the system guarantees SEC, as described in § 2.1 and (2) the system follows the semantics of the Add-Wins Set, *i.e.* in the case of a concurrent add and remove into the set of the same element, the element will be added. Since later experiments tests all implementations described in § 5.3, thus, correctness had to be validated for every implementation.

We designed an experiment that tested both of these properties. The test creates 1000 operation from each node, each of them concurrent to each other. This is achieved by toggling each node offline before creating the operations. One node only creates insert operations, whereas all other create remove operations. We then toggle all nodes online again and as soon as all operations have been delivered, we verify at all nodes, that the Add-Wins Set evaluate to a set containing the numbers 1 to 1000. The test can be described as the following in pseudo-code:

```
for each node in nodes:
  node.setOffline(true)

for each node in nodes:
  if node == nodes.first:
    node.AddWinsSet.insert(1 ... 1000)
  else:
    node.AddWinsSet.remove(1 ... 1000)

for each node in nodes:
  node.setOffline(false)

wait until all operations have been delivered

for each node in nodes:
  expect(node.AddWinsSet == 1 ... 1000)
```

This experiment tests property (1), since it makes sure that all nodes have an equivalent state and it tests property (2) since we require every add operation to "win" over the concurrent remove operations.

We then ran the experiment on a network of 4 nodes, over all TCSB implementations 100 times, to guarantee all of them has a correct implementation. The results of this are shown in Fig. 6.1, and indeed, judging by the results from this experiment it seems that all implementations both provide SEC and respects the Add-Wins Set semantics. From this we can conclude that our first hypothesis—that the pure CRDT framework by Baquero et al. [2017] and the eager garbage collection by Bauwens and Gonzalez Boix [2019] can be adjusted to work in a P2P Web environment—holds true.

Figure 6.1: The results of the correctness validation test across different TCSB implementations. The test was run 100 times.

Implementation	**Provides SEC?**	**Respects Add-Wins Set Semantics?**
OTCSB	Yes	Yes
ETCSB	Yes	Yes
NTCSB	Yes	Yes

6.2 Memory Usage Between Implementations

To prove or disprove our second hypothesis—that the eager garbage collection is equally effective in the Web environment as it was in LuAT—we wanted to reproduce the experiments by Bauwens and Gonzalez Boix [2019]. One of these experiments tested the memory usage across the different TCSB listed in § 5.3. As mentioned in § 3.3, absolute memory usage comparison is not the most relevant metric, but rather relative memory usage and relative memory growth is a more fair comparison. However, we will present the total memory usage results at first in order to analyse the memory usage behaviour over time.

In this experiment, Bauwens and Gonzalez Boix [2019] repeatedly told one node in a full-mesh peer-to-peer network to insert an integer number between 1 and 100 into an Add-Wins Set shared by the network—in total, they performed 1000 insertions and every 100 insertion, they changed the designated source node. The results from their experiment has been attached in Fig. 6.2. As is visible in the figure, Bauwens and Gonzalez Boix [2019] found the ETCSB implementation to be more effective than the traditional OTCSB, and even more effective compared to NTCSB. Indeed, meta-data could be removed continuously at all replicas, instead of having to delay removal until all nodes had transmitted causal information, which results in the saw-tooth pattern in their results.

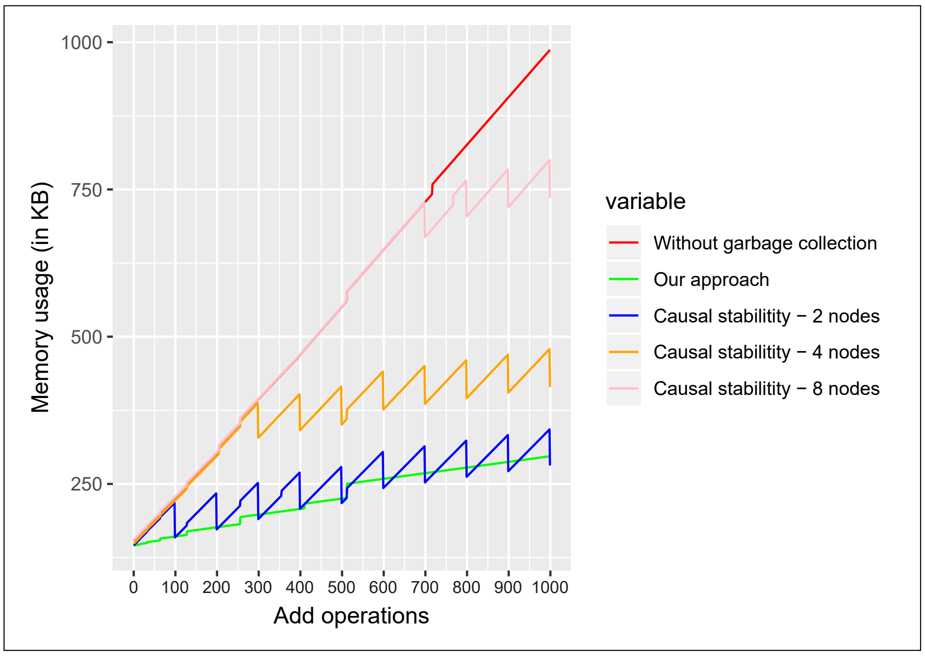


Figure 6.2: The LuAT results from the memory usage comparison test [Bauwens and Gonzalez Boix, 2019, Figure 7]. "Without garbage collection" corresponds to NTCSB, "Our approach" to ETCSB and "Causal Stability" to OTCSB. Permission has been granted from the authors to attach this figure.

In this experiment, as described by Bauwens and Gonzalez Boix [2019], causal redundancy would prevent memory from growing significantly after 100 operations, since then the same numbers would be repeatedly inserted. Hence, as established in § 5.4.2, we tweaked this experiment in our reproduction so that causal redundancy is not triggered. Instead of repeatedly inserting the numbers 1–100, we inserted only unique numbers between 1–1000, in a linearly increasing fashion.

Just as Bauwens and Gonzalez Boix [2019] did, we ran this test across five different network configurations and implementations: OTCSB with 2, 4 and 8 nodes, ETCSB with 4 nodes, and NTCSB with four nodes. We compared the absolute memory usage, the relative memory usage and the relative memory growth between the Web and LuAT implementations.

6.2.1 Absolute Memory Usage

The absolute memory usage of this experiment is shown in Fig. 6.3, and shows the mean and standard deviation across 20 runs for the first node, both measured in

(A) as total heap usage and (B) as serialised JSON string length. The low standard deviation across all implementation in (A) tells us that the measurements are stable and reproducible, meaning we can trust this is the results we will consistently get when running the experiment. Only the memory usage from one node is shown in the figure as the results did not differ between different nodes.

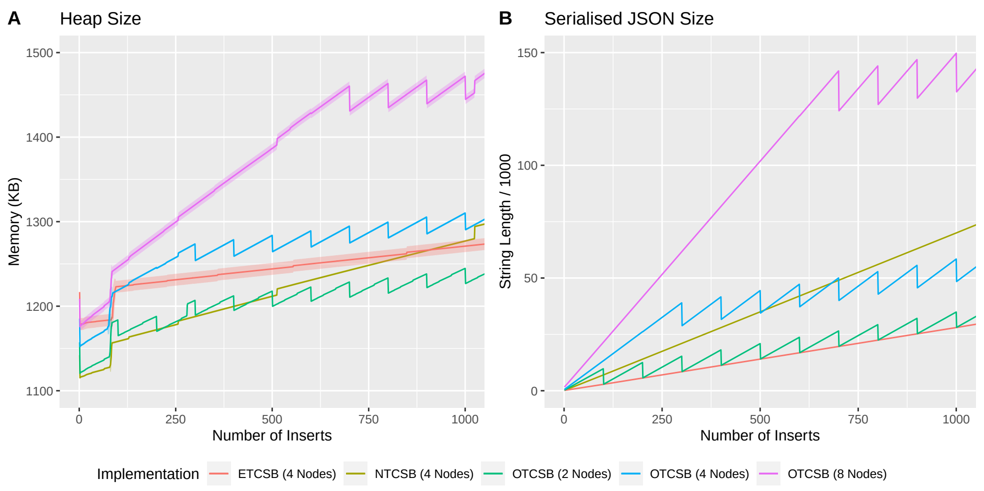


Figure 6.3: Comparison of memory usage between different implementations of the TCSB. The figure shows the mean and standard deviation over 20 runs as a solid line and shaded area around this line, respectively.

Our first observation of the results in Fig. 6.3 is that the slope of the data between (A) and (B) correspond with each other, while initial memory usage differ in (A). This is most likely caused by two different things, (1) is that (A) will measure the memory usage of the internal WebRTC connections, while (B) does not, resulting in the configurations with more connections to use more memory, and (2), different implementations of the TCSB is differently affected by the stabilisation phase, which is to be expected since they all use different internal data structures that the engine optimise differently. We also observe an anomaly around the 100th insert for all configurations, similar to the anomaly in § 5.4.2—as previously stated, this is most likely caused by the engine resizing a container prematurely as an optimisation, considering it happens for all configurations at the same time. Nevertheless, from these observations we can confirm that (A) reflects the objects graph size shown in (B), which gives us the confidence that the measurements are accurate.

Our second observation is about the different behaviour across the different implementations. The OTCSB implementations must wait for causal information from all nodes before it can perform any causal stabilisation, which means that it has to wait for 100, 300 and 700 operations for a network of 2, 4 and 8 nodes, respectively; after this it can perform some PO-Log compaction after every 100 operations, as the designated source node changed and provided more causal stability information. We can see this in Fig. 6.3 as a saw-tooth pattern. The ETCSB, however, continuously perform some causal stabilisation, which simply results in a line with low slope, especially compared to NTCSB that performs no causal stabilisation at all, which results in a much steeper slope. This is the exact same behaviour observed in the LuAT implementation in Fig. 6.2, which gives us confidence that both our experiment and implementation is comparable to theirs.

This, however, is where the similarities between our and Bauwens' and Gonzalez Boix's [2019] results end; the first difference being that in Fig. 6.3, different configurations of OTCSB have a different initial slope, while in Fig. 6.2, all different configurations of OTCSB have an identical initial slope. However, according to the theory presented in § 2.3 and § 2.6.2, we would expect different slopes. The reason for this is that the slopes of the OTCSB configurations with 2, 4 and 8 nodes should differ, since the vector clocks from a network with many nodes uses more memory to store than a vector clock from a network with fewer nodes, and with every non-stable operation we have to store at least one additional vector clock. This is represented in both (A) and (B) of Fig. 6.3, but not in the LuAT results in Fig. 6.2. This discrepancy could possibly be explained due to differences between the Web implementation and the LuAT implementation. Indeed, there is a possibility that the LuAT implementation uses a fixed size vector clock—indeed, if we force our vector clocks to always be the same fixed size of 8, we get the same behaviour, as can be sen in Fig. 6.4. Using a fixed size vector clock is a waste of memory, which means we would not want to do this in an actual implementation. However, there is a possibility that the reason for why the clock sizes seem to be fixed in LuAT could be due to the growth behaviour of Lua tables. Namely, Lua tables seems to have space pre-allocated and double in size when it needs to grow[1], something that may very well be the cause for this behaviour.

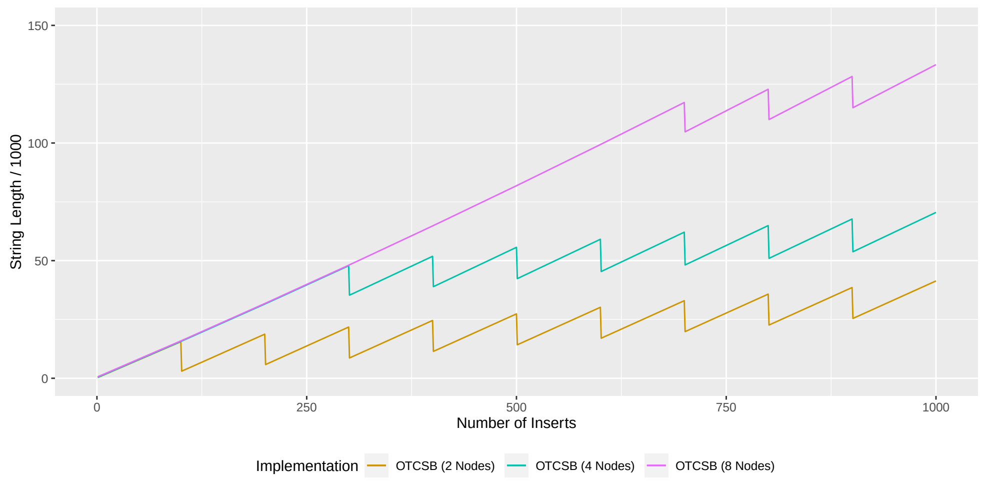


Figure 6.4: The serialised JSON size when forcing clocks to be a fixed size of 8, regardless of the actual network size.

A second difference between the Web and LuAT implentation, is that in Fig. 6.3, the NTCSB (4 Nodes) has a lower initial slope than that of OTCSB (4 Nodes), while in Fig. 6.2, they have an identical initial slope. In our implementation, we both expect and observe that NTCSB (4 Nodes) has an initially lower slope than that of OTCSB (4 Nodes), since it does not have to perform the necessary bookkeeping to decide causal stability, *i.e.* store a list of timestamps that has yet to become causally stable. The reason for this discrepancy could, once again be due to differing implementations between the Web and LuAT. Namely, in our Web implementation, a list of causally unstable timestamps are stored both in the OTCSB and the PO-Log. If the OTCSB implementation had knowledge of the PO-Log, we could prevent this duplication,

[1]`http://www.lua.org/source/5.3/ltable.c.html`

causing NTCSB (4 Nodes) and OTCSB (4 Nodes) to have an identical initial slope. It is possible that the LuAT implementation by Bauwens and Gonzalez Boix [2019] has made such an optimisation. Conversely, it is also possible that the NTCSB implementation by Bauwens and Gonzalez Boix [2019] is the same as OTCSB, *i.e.* keeping track of causal stability, but never actually calling the stable hook.

One last difference between the Web and LuAT results is whether the ETCSB is always the most memory efficient implementation for a network. Indeed, as can be seen in Fig. 6.3 (A), ETCSB is only more efficient than NTCSB after almost 800 unique insert operations, for a network of 4 nodes—whereas ETCSB is the most efficient implementation from the start in LuAT. In Fig. 6.3 (B), however, we see a behaviour that more closely resembles that of LuAT in Fig. 6.2, which means that the gap is most likely a result of our stabilisation phase, whose effect is not visible in the serialised JSON size measurements—indeed, if the ETCSB line in (A) started with the same memory usage as NTCSB, it would be more efficient from the start. Furthermore, the NTCSB on the Web also has an initially lower memory usage than OTCSB for a network of 4 nodes until after 1000 and 500 insert operations in (A) and (B) respectively, something that is not observed in LuAT. This is due to the same reason as previous discrepancies, that the stabilisation phase affects the implementations differently. In the long-run, however, given that OTCSB can continue to perform causal stabilisation, it is more efficient. This is evident in Fig. 6.5 which shows the same test, but for 10000 inserts, instead of 1000. Figure Fig. 6.5 also makes it clear that the longer a system runs, the more memory will be saved by using the causal stabilisation PO-Log compaction technique, while for a system that often resets its state—as the stabilisation phase in § 5.4.1 did—it might be more efficient to not use any method of garbage collection at all to avoid the cost of engine pre-allocations.

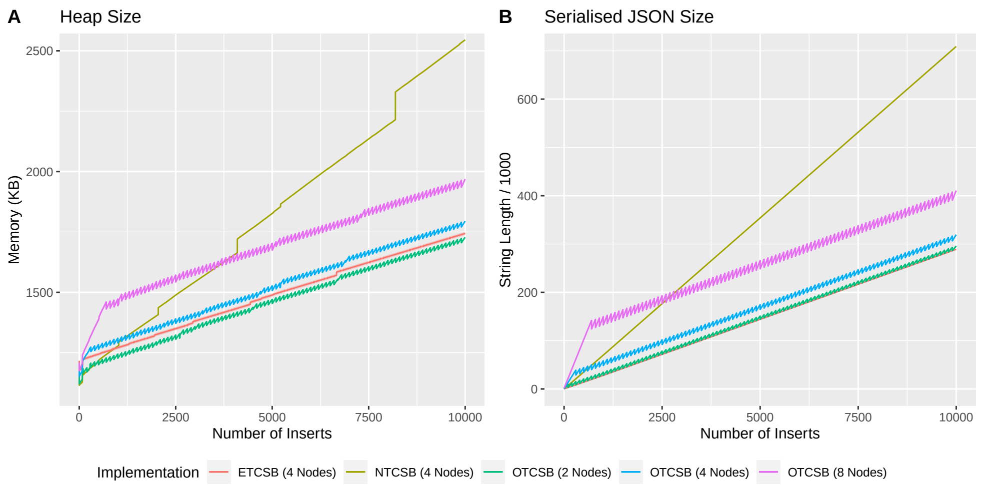


Figure 6.5: Comparison of memory usage between different implementations of the TCSB. The figure shows the mean and standard deviation over 20 runs as a solid line and shaded area around this line, respectively.

6.2.2 Relative Memory Usage

Our second hypothesis—that the eager garbage collection is equally effective in this new environment—was regards to relative memory usage, while both Fig. 6.3

Fig. 6.5 show absolute memory usage. To generate the relative memory usage we can divide each line in Fig. 6.3 by a baseline. In our case, the NTCSB (4 Nodes) is an appropriate baseline as that is a least effort implementation, *i.e.* doing nothing at all. The relative memory usage on the heap, compared to the configuration NTCSB (4 Nodes) is shown in Fig. 6.6 (A), next to the relative memory usage of the same implementations in LuAT shown in Fig. 6.6 (B), based on an approximation of the results in Fig. 6.2—it is only an approximation as we only have access to the figure, not the raw data, thus, the saw-tooths visible is not represented in the approximation. We have chosen to only include an analysis of relative memory usage of the heap size, and not the serialised object graph size; while the serialised object graph size is an interesting metric, it is not a true representation of memory usage and should thus not be used for comparisons between the Web and LuAT implementations.

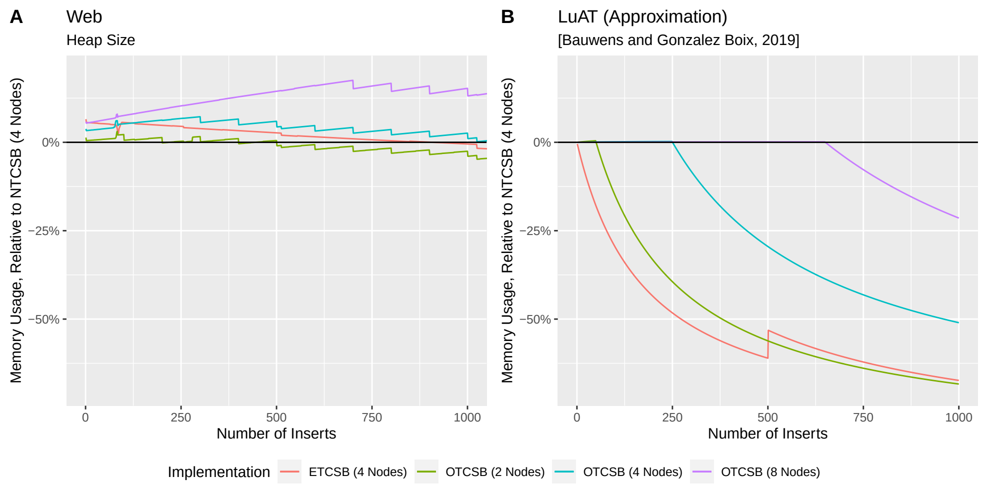


Figure 6.6: Comparison of the relative memory usage of different implementations compared to NTCSB (4 Nodes), running on a Web browser for 1000 operations. (A) is generated from a mean of 20 runs, while (B) is an approximation of the data presented by Bauwens and Gonzalez Boix [2019]

In Fig. 6.6 we note two clear differences between (A) and (B)—the first being that in (A), all implementations are initially worse than NTCSB, while in (B), implementations uses the same, or less, memory than NTCSB. This is expected for the same reasons as previously discussed, partly due to the stabilisation phase affecting the configurations differently and partly due to the bookkeeping that the OTCSB implementations requires when deciding causal stability, something that the NTCSB does not have to perform. The second difference between (A) and (B) is that after 1000 operations, all implementations in (B) are greatly more memory efficient than NTCSB.

However, as was obvious from Fig. 6.5 is that the slopes between the implementations were drastically different, which means that after the system had run for 10000 operations, the offset caused by the stabilisation phase had diminished. If we generate the same relative memory usage over 10000 operations, we get Fig. 6.7. Here we can see that all configurations of a network are more efficient than NTCSB (4 Nodes), which is to be expected from previous observations. We can also recognise that ETCSB (4 Nodes) are consistently more efficient than OTCSB (4 Nodes), similarly to the findings of Bauwens and Gonzalez Boix [2019].

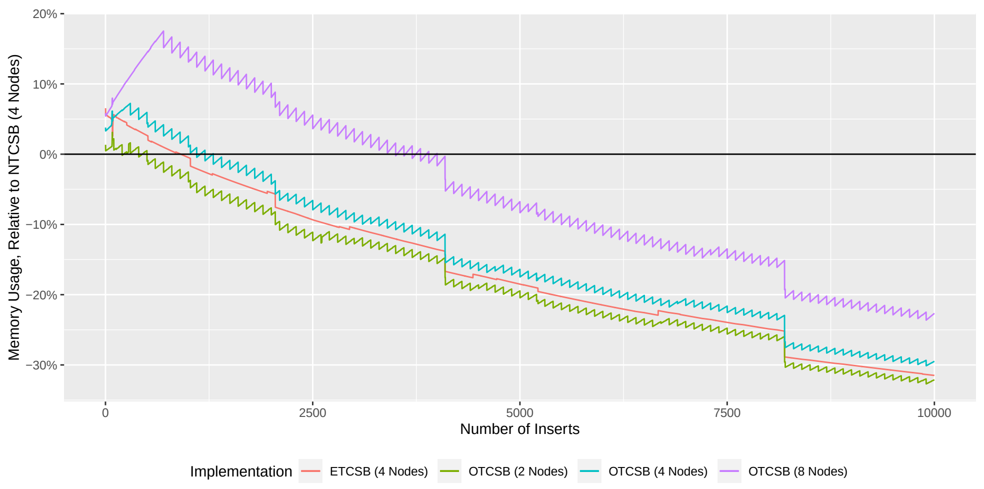


Figure 6.7: The relative memory usage of different implementations running compared to NTCSB (4 Nodes), running on a Web browser for 10000 operations. The data is generated from the mean of 20 runs.

However, despite running the test for 10000 operations, the Web implementation of the CRDT framework does not reach the same relative memory usage improvements as in the experiments performed in LuAT by Bauwens and Gonzalez Boix [2019]. Indeed, the ETCSB implementation in Fig. 6.6 (B) reaches the same relative memory usage improvements after only 100 operations, as the ETCSB implementation in Fig. 6.7 does in 10000 operations. This is partly due to the same reason as previous observations—stabilisation effects and possibly differing implementations—but it is also a result from runtime differences; the JavaScript VM has a larger initial memory footprint than that of Lua, which diminishes any relative memory usage improvements in the beginning of the system's lifetime.

This leads us to our last comparison—comparing the memory growth across implementations and between running on the Web vs LuAT. By comparing the memory growth, we remove any side effects produced by the stabilisation phase or runtime differences. Furthermore, since we compare the relative growth, any differences between the Web and LuAT is removed.

6.2.3 Relative Memory Usage

The relative memory growth can be obtained by simply measuring the slope of a line from the start of the measurement until after a certain operation in Fig. 6.3, and then dividing this by a baseline. Since this measurement does not rely on the start value, only by its growth, it removes any side effects produced by the stabilisation phase or runtime differences. Once again, we have chosen NTCSB (4 Nodes) as the baseline for the same reasons as when comparing the relative memory usage. The relative memory growth can be seen in Fig. 6.8, where (A) is a direct calculation of the result presented in Fig. 6.5 and (B) is an approximation of the results presented by Bauwens and Gonzalez Boix [2019]. (A) shows the growth after 100, 1000, 5000 and 10000 operations, while (B) shows the growth after only 100 and 1000 operations, since Bauwens and Gonzalez Boix [2019] only ran the experiment for 1000 operations.

Here we can yet again observe some similarities between Fig. 6.8 (A) and Fig. 6.8 (B).

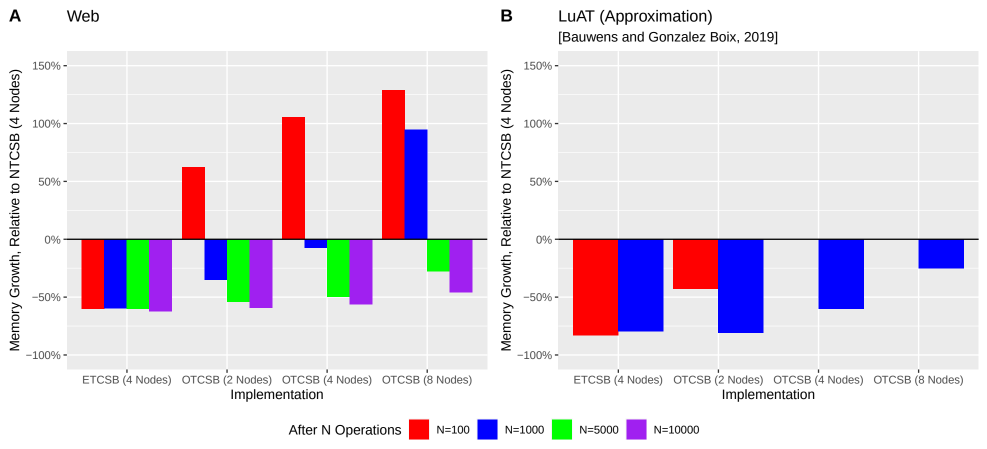


Figure 6.8: The relative memory growth after N operations compared to NTCSB (4 Nodes), running on the Web (A) and an approximation of the LuAT results presented by Bauwens and Gonzalez Boix [2019].

The first similarity is that ETCSB grows at a lesser rater than NTCSB in both (A) and (B), and the second similarity is that this does not change significantly over time. This is to be expected as NTCSB and ETCSB grows linearly in both our results and the results of Bauwens and Gonzalez Boix [2019]. For the OTCSB configurations, however, we notice in (A) and (B) that the more elements we insert, the greater the difference. This is also to be expected from looking at *e.g.* Fig. 6.5, as the OTCSB configurations have to initially wait for operations from all nodes before it can start its causal stabilisation PO-Log compaction, eventually leading to a less steep slope. NTCSB, on the other hand, will always have the same steep slope from the start.

We can then observe the differences between Fig. 6.8 (A) and Fig. 6.8 (B). The first being that the OTCSB configurations have an initial steeper growth than NTCSB in (A), while this is not the case for (B). This was also observed when discussing Fig. 6.3 (B), and most likely due to differing implementations, namely that the Web implementation have duplication of data that Bauwens and Gonzalez Boix [2019] are able to avoid. The second difference is that in (B) we observe that all implementations grow less fast compared to NTCSB than in (A). This may, once again, be due to the Web and Lua implementations being different in some regard.

In conclusions, this experiment has surfaced two main differences between our implementation on the Web versus the results on LuAT by Bauwens and Gonzalez Boix [2019]. The first difference is that, while individual configurations on the Web have identical behaviour to that of LuAT, their collective behaviour differs—this is especially evident in Fig. 6.3 (B), where the NTCSB and OTCSB configurations all have a different initial slope, opposed to the having the same initial slope as in the results presented by Bauwens and Gonzalez Boix [2019]. As previously discussed, our results match our intuition: for OTCSB, different amount of nodes should result in differing slopes due to differently size of vector clocks, and NTCSB (4 Nodes) should have a less steep slope than OTCSB (4 Nodes) since it does not require any bookkeeping. Our only guess as to why this differs are due to different implementations, where the NTCSB in LuAT seems to grow faster than its counterpart on the Web. The second main difference is that on the Web, the ETCSB and OTCSB implementations underperforms its LuAT counterparts. It is impossible to say whether this is due to implementations being less performant on the Web, or because of our previous guess—that the NTCSB implementation is less efficient in the LuAT environment—since NTCSB is our common baseline, this would skew the results in favour of LuAT. Regardless, we can conclude that ETCSB is equally effective, if not more effective, than the traditional OTCSB approach, with the additional benefit of not having to wait on other nodes before it is able to perform causal stabilisation, which was the main benefit touted by Bauwens and Gonzalez Boix [2019]. Additionally, we found that applying garbage collection may only be beneficial if the state is never reset, due to stabilisation effects.

6.3 Temporary Outage Behaviour

The second experiment performed by Bauwens and Gonzalez Boix [2019] tested the efficiency of ETCSB in the event of temporary outages, compared to that of NTCSB, in their LuAT implementation. The purpose of reproducing this experiment is similar to that of the previous experiment—namely to gather data to either prove or disprove our second hypothesis. However, as one core feature of CRDTs is that they remain highly available, even in the event of network partitions, we also want to investigate if our implementation on the Web is equally effective as the LuAT implementation, during these temporary outages.

The experiment started by inserting the same number 100 times, starting from number 1. After 100 insertions of the same number, they toggled the connectivity status from online to offline, and moved to the next number, *i.e.* the number 2. They then repeated this procedure of inserting the same number 100 times and toggling the connectivity status. After every operation, they measured the size of the heap. The results from their experiment, which has been attached in Fig. 6.9, tested both the NTCSB and ETCSB implementations, and showed that for ETCSB, memory usage spiked during the temporary outage due to not being able to share causal stability information during this offline period, and thus not being able to perform any metadata removal. When the node reconnected, memory usage decreased to a normal level, well below NTCSB. Conversely, NTCSB not affected at all by the temporary outage, since there is no causal stabilisation procedure that could be affected by it.

The test performed by Bauwens and Gonzalez Boix [2019] is unfortunately difficult to accurately reproduce due to two inconsistencies. The first inconsistency is that NTCSB was completely unaffected, and did not show any increase during the offline period, not even due to operations that are buffered during the offline period, which they were according to the system model described by Bauwens and Gonzalez Boix [2019]. This leads to the conclusion that the buffered operations was not included in the measurements by Bauwens and Gonzalez Boix [2019]. However, our course grained method of measuring the size of objects allocated on the heap, described in § 5.1, will include the buffered operations in its measurements, causing a discrepancy between our results and the results of Bauwens and Gonzalez Boix [2019]. This would lead to an unfair comparison during the offline period of the test, but luckily, it would still be comparable during the online period. Fortunately, for the serialised object graph method of measuring memory, we can selectively choose what to include in the serialisation. Consequently, we chose to not include the buffered operations in the serialised object graph to provide as similar results to that of Bauwens and Gonzalez Boix [2019] as possible.

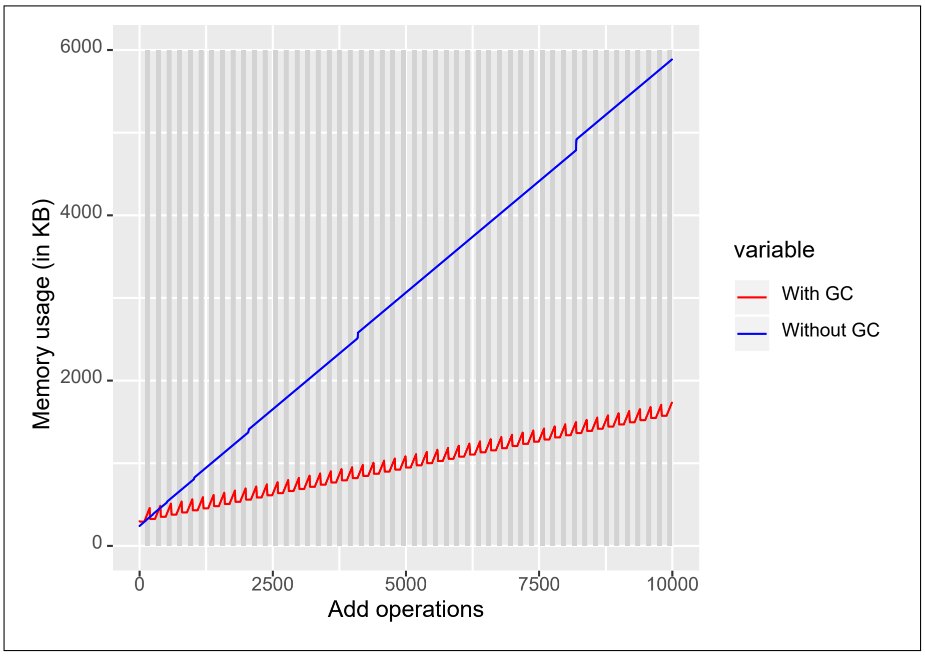


Figure 6.9: The results from LuAT when testing the temporary outage behaviour [Bauwens and Gonzalez Boix, 2019, Figure 8]. "With GC" corresponds to ETCSB and "Without GC" to NTCSB. Permission has been granted from the authors to attach this figure.

The second inconsistency is, once again, that if the same operation would be repeatedly sent 100 times, the graph should not increase linearly, but only when a new number is added. In the results of Bauwens and Gonzalez Boix [2019], however, this is not the case, as their results are linearly increasing. As stated in § 5.4.2, we will thus adjust the experiment so that no causal redundancy is performed so that a comparison between our results and the results of Bauwens and Gonzalez Boix [2019] can be made. We chose to use the same approach from the last experiment by linearly increasing the number we send, *i.e.* sending the numbers 1–10000, as this would grant us an interesting comparison between the behaviour when there are no outages, which is the previous experiment, and the behaviour when there are temporary outages, which is this experiment.

The measured absolute memory usage during this experiment can be seen in Fig. 6.10 and shows the heap usage in (A) and the size of the serialised object graph in (B). Indeed, as predicted, (B) is more similar appearance-wise to Fig. 6.9 than (A) is, most likely due to it not including the buffered operations. Regardless, both (A) and (B) exhibits the same behaviour noted by Bauwens and Gonzalez Boix [2019], that for ETCSB, we see a spike in memory usage during the offline period due to not being able to perform any causal stability PO-Log compaction, while NTCSB is largely unaffected, except for the buffered operations in (A).

The total memory usage shown in Fig. 6.10 gives us a rough idea that the Web implementation and the LuAT implementation behaves similarly in the event of outages, except the spikes for NTCSB (4 Nodes) in (A). One last observation is that the results in Fig. 6.10 coincides with the results from our previous experiment, presented in Fig. 6.5. Analysis of relative memory usage and growth was also performed, but

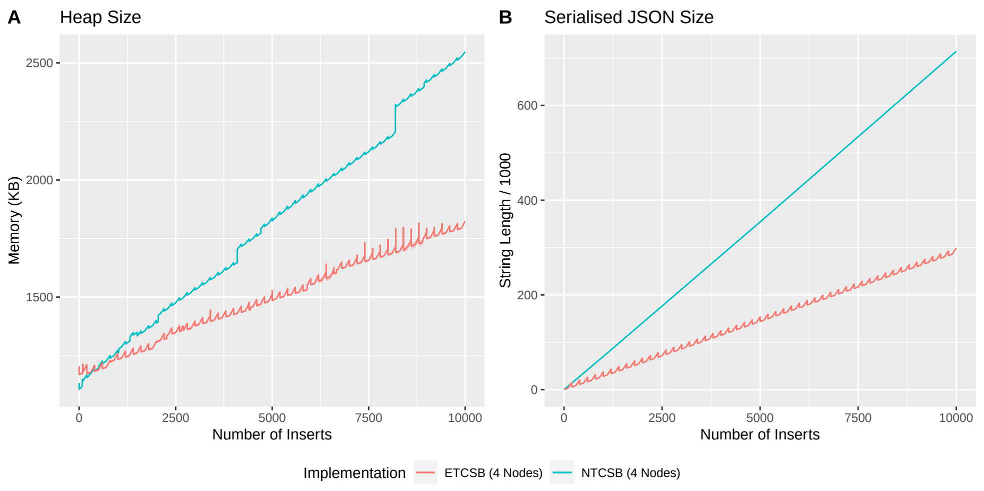


Figure 6.10: The behaviour of ETCSB and NTCSB during temporary outages, both shown in heap usage (A) and the size of the serialised object graph (B). The line shows an average over 20 runs and the shaded area represents the standard deviation.

they too coincided with the previous findings in Fig. 6.7 and Fig. 6.8 respectively, and will thus not be presented here.

From this experiment we can conclude that partitioning the network did not have an effect on total memory usage once network was joined again, *i.e.* memory usage corrected itself after the temporary outage. These findings do not help us prove our second hypothesis—that the eager garbage collections is equally effective on the Web as in LuAT—but neither does it disprove it, and as an additional benefit, it strengthens the case of a correct implementation that provides SEC.

6.4 Validity

Since the environment of the Web and LuAT are different, we had to make choices when reproducing these experiment that may, or may not, threaten the validity of the results presented in this chapter. We will therefore discuss any such issues here and their implications, if any.

First is the issue of only testing one Web browser, after all, one of the guiding principles of W3C, as mentioned in Chapter 1, is that the Web should run on everything [w3c, 2017]. It is impossible to know if our implementation work correctly, and is equally memory efficient, in other Web browsers without running the same test suites with these browsers, instead of Chrome. We can however make an educated guess that our implementation works across all Web browsers, since the implementation is built on the open standards governed by the W3C, *e.g.* WebRTC. This means, given that another Web browser has a correct implementation of WebRTC, our implementation should work for that browser as well. It is however possible that memory usage behaviour might differ between different Web browsers, especially those that use different garbage collection techniques than V8—the JavaScript VM used by Chrome. However, we consider testing our implementation in V8 to be the most important, since it has the highest market share on the Web, with more than

60% market share [StatCounter, 2019]. V8 is also the JavaScript VM used by the JavaScript runtime Node.js[2], which further bolsters the argument that V8 is the most important target for testing, if we are forced to choose. At last, while memory usage behaviour might differ, *e.g.* another engine might not require the same stabilisation phase as V8 to get stable results, the relative memory and growth should regardless be similar between the engines.

Another threat to validity are the changes we made to the experiments, especially with regard to causal redundancy as described in § 2.6.1. Since it was not possible to determine exactly how Bauwens and Gonzalez Boix [2019] performed their experiments, we had to guess based on knowledge from our background research. Hence, there exists a possibility that we have tested different behaviour of the algorithm than Bauwens and Gonzalez Boix [2019]. We do, however, consider this to only be a slight risk, since we performed experiments to validate some of our guesses and since the behaviour of our algorithm was shown to be similar that of Bauwens and Gonzalez Boix [2019] across both replicated experiments.

One last threat to validity is the stabilisation phase explained in § 5.4.1, which we later use to explain several anomalies in § 6.2; if the stabilisation phase causes anomalies in the testing, why have it in the first place? The answer to this is that it was necessary to perform a stabilisation phase before experiments to produce accurate results. Indeed, as shown in § 5.4.1, our memory measurements grew for the first 15 stabilisation rounds while we did not expect it to. Without using a stabilisation phase, we observed both more frequent and more severe anomalies, which could only be explained due to engine optimisations—these disappeared after the stabilisation phase, which made us prefer these results. A stabilisation phase also more accurately represents a long-running system, as opposed to a short-running one, as the system would "naturally" stabilise after it had run for a while.

Another question related to the stabilisation phase is: why do we require a stabilisation phase if Bauwens and Gonzalez Boix [2019] did not have one? Is it a fair comparison? Indeed, Bauwens and Gonzalez Boix [2019] mentions nothing of a stabilisation phase, but their results did not show any of the anomalies experienced during our experiments. The most likely explanation is that the Lua environment used by Bauwens and Gonzalez Boix [2019] would behave the same in a short-running system as a long-running one, which would make a stabilisation phase superfluous. Due to this, we consider comparing our results on the Web with a stabilisation phase to the LuAT results to be more accurate than not having a stabilisation phase at all, since we would not compare the memory usage by the algorithm, but rather the underlying engine optimisations.

[2]`https://nodejs.org/`

Chapter 7

Conclusions

The purpose of this work was to investigate the promising family of data types called conflict-free replicated data types, or CRDTs, in a peer-to-peer distributed system. CRDTs provides a specific type of consistency called Strong Eventual Consistency (SEC) by relying on commutativity [Shapiro et al., 2011], and thus have number of useful applications—both those hidden from end-users, and those who put the conflict-free experience in focus, *e.g.* real-time collaborative applications. These types of applications rely on data that is omnipresent, a term derived from Baldemair *et al.*'s [2013] vision of the future, to have data that is available "anywhere and anytime to anyone and anything".

Furthermore, we noted that many real-time collaborative applications exists on the Web, but none that we found makes use of peer-to-peer collaboration—something we blame the lack of appropriate APIs for. However, the API area of the Web has lately evolved, and now provides means to establish peer-to-peer connections via WebRTC [WebRTC Specification]. The Web is thus a suitable target platform for CRDTs, and together with the inclusive principles of the World Wide Web Consortium (W3C)—Web on Everything and Web for All [w3c, 2017]—CRDTs on the Web in not only compatible with Baldemair *et al.*'s [2013] future, but may one day be part of realising it.

A problem for many CRDTs is that they have to store ordering information to ensure SEC, and if no schemes for removing meta-data are employed, memory use may grow without bound [Bauwens and Gonzalez Boix, 2019]—something that goes against the W3C's principle of Web on Everything and Web for All, since there exists devices that runs the Web that does not have memory to spare, nor does everyone have the economical resources to purchase devices with larger memory. Our purpose was thus to provide memory efficient, highly available and strongly eventual consistent data, shared through peer-to-peer communication via the Web that upholds the W3C's values.

Previous work by Bauwens and Gonzalez Boix [2019] proposed and implemented an eager garbage collection scheme that could be used to quickly identify and remove the aforementioned meta-data—this was implemented in Lua. The eager garbage collection is an extension of the work by Baquero et al. [2017], who proposed a memory efficient general framework for designing CRDTs. Thus, a suitable goal to help realise this work's purpose was to re-implement the general CRDT framework by Baquero et al. [2017] on the Web and adapt the eager garbage collection extension by Bauwens and Gonzalez Boix [2019] to work in this new environment. To establish whether our goal was met or not, we needed to evaluate if the CRDT framework could be adapted to run P2P on the Web and if similar memory improvements from the eager garbage collection scheme could be found in this new environment. To this end, we turned to the scientific tradition of reproduction—namely to reproduce the work of Bauwens and Gonzalez Boix [2019].

To help us decide whether we reached our goals, we declared two hypotheses: (1) that the general CRDT framework could be adjusted to work in a P2P Web environment, and (2) that the eager garbage collection scheme by Bauwens and Gonzalez Boix [2019] is equally effective in this environment. The evaluation criteria for (1) is that our implementation is correct, *i.e.* provides SEC, and for (2), that the same relative memory savings could be achieved on the Web as in Lua.

To evaluate our first hypothesis, we stress tested a specific type of CRDT called the Add-Wins Set, by creating many concurrent operations to see if all replicas of the set converged to the expected results according to the semantics of the Add-Wins Set. Fortunately, we could confirm the correctness of our implementation according to these criteria and could conclude that the general CRDT framework indeed could be adjusted to work in a P2P Web environment.

To evaluate our second hypothesis we reproduced the same experiments performed by Bauwens and Gonzalez Boix [2019], but in our Web implementation, instead of Lua, and then later compared our results with theirs. In doing this, we noticed some discrepancies between our results and the results of Bauwens and Gonzalez Boix [2019], most likely caused by a non-comparable baseline due to insufficient experiment descriptions by Bauwens and Gonzalez Boix [2019]. Even worse, due to troubles accurately measuring memory on the Web and inconsistencies in the original experiment descriptions, changes to the experiments had to be made which scarifies our results validity, as mentioned in § 6.4.

Regardless, the conclusions from our experiments showed that different garbage collection schemes behaviour was mostly the same for both the Web and Lua, but Lua seemed to consistently outperform its corresponding Web implementations in all experiments and metrics, *e.g.* , the eager garbage collection on the Web grew 0.4 as fast as no garbage collection at all, compared to 0.2 in Lua. Whether this is due to differences in the environment, the changes made to the experiments or due to a non-comparable baseline is impossible to say without more information about the original experiment. To summarise, while we did not find enough evidence to prove our second hypothesis, we can not disprove it, either. Nevertheless, the same benefits of eager garbage collection were found to also apply to the Web environment, even if the relative memory savings were less. An additional finding was that whether to apply garbage collection at all on the Web may only be worth it in systems that seldom reset its state, since engine memory stabilisation effects will render any memory savings directly after a reset insignificant.

This work proves two main points: the first is that both the CRDT framework by Baquero et al. [2017] and the eager garbage collection by Bauwens and Gonzalez Boix [2019] is well-suited for the Web, and the second is that CRDT seems to live up to their promised potential. Thus, this research is interesting, since not only does it prove that previous CRDT research can be applied in different environments than they were initially designed for, but it shows promise for the Web as a viable platform to the future. Furthermore, this work is not only research, but a prototypical implementation of a general CRDT framework, which can be used as a building block for many CRDTs, that will automatically receive the benefit of the garbage collection techniques investigated in this report.

As previously discussed, CRDTs shows great promise, *e.g.* it could be part of realising Baldemair *et al.*'s [2013] vision of having data available "anywhere and anytime to anyone and anything". This thesis, however, does not discuss how CRDTs could be used to achieve this future, but rather investigates CRDTs potential on the Web. Moreover, there exists more work to be done for the prototypical implementation described in this thesis for it to be considered useful—only then can an investigation be made if CRDTs on the Web can be part of Baldemair *et al.*'s future. With this in mind, we have identified three points of future work that all would work towards this future.

The first point would be to extend our static network to a dynamic one. Bauwens and Gonzalez Boix [2019] *e.g.* suggests a dynamic join model that would allow peers to join the network at any time, and for the system to keep its SEC. This could be further extended with dynamic leaving of the network, both announced and unannounced—this is a must for real-world applications on the Web, where users expect to be able to enter and leave at will.

The second point of future work is to perform a similar set of tests for another CRDT than the Add-Wins Set; the Add-Wins Set is a simple CRDT and have limited use *e.g.* when building near real-time collaborative applications. For example, it would be interesting to investigate if current collaborative text editing CRDTs, such as RGA [Roh et al., 2011] could be adapted to work in Baquero *et al.*'s [2017] general CRDT framework, without sacrificing algorithmic time complexity. To this end, the same benchmarking and testing suite used in this work could be extended to also measure execution time when generating and applying operations.

Our third and last suggestion of future work is to experiment with different network layers. Indeed, WebRTC the only choice for P2P communication over internet on the Web, but lately work has been done to draft a specification of the WebAPIs WebUSB[1] and Web Bluetooth[2], two protocols that allows communication over USB and Bluetooth respectively. Namely, it would be interesting to know if the same general CRDT framework could be adapted to work over networks that uses USB and Bluetooth as its underlying connection.

It is evident that CRDTs on the Web shows great potential, but that there is still a lot of work to be done. Despite this, we should not dismiss this work nor other current research on CRDTs, a tool that may very well help us reach a future where data is omnipresent. Indeed, as more research and real-life implementations develops, the closer we are to a world where data is available anywhere and anytime. In this work we have explored one possible direction of realising this on the Web, but there is still a long way to go, and we are excited for possible future inventions in this area.

[1] `https://wicg.github.io/webusb/#enumeration`

[2] `https://webbluetoothcg.github.io/web-bluetooth/`

Bibliography

W3C Mission, 2017. URL `https://www.w3.org/Consortium/mission`. (accessed 2020-05-28).

Robert Baldemair, Erik Dahlman, Gabor Fodor, Gunnar Mildh, Stefan Parkvall, Yngve Selen, Hugo Tullberg, and Kumar Balachandran. Evolving wireless communications: Addressing the challenges and expectations of the future. *IEEE Vehicular Technology Magazine*, 8(1):24–30, 2013.

Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Pure operation-based replicated data types. *arXiv preprint arXiv:1710.04469*, 2017.

Jim Bauwens. Personal Communication, 2020.

Jim Bauwens and Elisa Gonzalez Boix. Memory efficient CRDTs in dynamic environments. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, pages 48–57. ACM, 2019.

Andrew Butterfield, Gerard Ekembe Ngondi, and Anne Kerr. CAP theorem, 2016. URL `https://www.oxfordreference.com/view/10.1093/acref/9780199688975.001.0001/acref-9780199688975-e-6615`.

Cisco Public. Cisco annual internet report - cisco annual internet report (2018–2023) white paper, Feb 2020. URL `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`. (accessed 2020-05-28).

George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed systems : concepts and design*. Pearson Education ; Addison-Wesley, Harlow, Essex, 5. ed., international ed. edition, 2012. ISBN 9780273760597.

Omar S Gómez, Natalia Juristo, and Sira Vegas. Replications types in experimental disciplines. In *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2010.

Peter Ivie and Douglas Thain. Reproducibility in scientific computing. *ACM Computing Surveys (CSUR)*, 51(3):1–36, 2018.

Lua. About, Sep 2018. URL `https://www.lua.org/about.html`. (accessed 2020-05-28).

Mozilla Developer Network. Javascript, Dec 2019a. URL `https://developer.mozilla.org/en-US/docs/Web/JavaScript`. (accessed 2020-05-28).

Mozilla Developer Network. WebAssembly Concepts, Dec 2019b. URL `https:`

//developer.mozilla.org/en-US/docs/WebAssembly/Concepts. (accessed 2020-05-28).

Mozilla Developer Network. WebRTC API, Oct 2019c. URL https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API. (accessed 2020-05-28).

Mozilla Developer Network. Signaling and video calling, Nov 2019d. URL https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling. (accessed 2020-05-28).

Mozilla Developer Network. Introduction to WebRTC protocols, Jul 2019e. URL https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols. (accessed 2020-05-28).

Mozilla Developer Network. Web APIs, Jan 2020f. URL https://developer.mozilla.org/en-US/docs/Web/API. (accessed 2020-05-28).

redislabs. Redis sentinel documentation, n.d. URL https://redis.io/topics/sentinel. (accessed 2020-05-26).

Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354 – 368, 2011. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2010.12.006. URL http://www.sciencedirect.com/science/article/pii/S0743731510002716.

Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24550-3.

StatCounter. Browser market share worldwide, Dec 2019. URL https://gs.statcounter.com/browser-market-share/all/worldwide/2019. (accessed 2020-05-28).

Jan Vitek and Tomas Kalibera. Repeatability, reproducibility and rigor in systems research. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 33–38. IEEE, 2011.

Werner Vogels. Eventually consistent. *Queue*, 6(6):14–19, October 2008. ISSN 1542-7730. doi: 10.1145/1466443.1466448. URL https://www.cs.princeton.edu/courses/archive/fall13/cos518/papers/ec-vogels.pdf.

WebRTC Recommendation. Webrtc 1.0: Real-time communication between browsers, Nov 2017. URL https://www.w3.org/TR/2017/CR-webrtc-20171102/. (accessed 2020-05-28).

WebRTC Specification. Webrtc 1.0: Real-time communication between browsers, Dec 2019. URL https://www.w3.org/TR/webrtc/. (accessed 2020-05-28).

WebSocket Protocol. The websocket protocol, Dec 2011. URL https://tools.ietf.org/html/rfc6455. (accessed 2020-05-28).