SHELFAWARE: ACCELERATING COLLABORATIVE AWARENESS WITH SHELF CRDT

A Thesis

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

John Waidhofer

March 2023

COMMITTEE MEMBERSHIP

TITLE: ShelfAware: Accelerating Collaborative Awareness with Shelf CRDT

AUTHOR: John Waidhofer

DATE SUBMITTED: March 2023

COMMITTEE CHAIR: Maria Pantoja, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Rodrigo Canaan, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Phoenix (Dongfeng) Fang, Ph.D.
Professor of Computer Science

ABSTRACT

ShelfAware: Accelerating Collaborative Awareness with Shelf CRDT

John Waidhofer

Collaboration has become a key feature of modern software, allowing teams to work together effectively in real-time while in different locations. In order for a user to communicate their intention to several distributed peers, computing devices must exchange high-frequency updates with transient metadata like mouse position, text range highlights, and temporary comments. Current peer-to-peer awareness solutions have high time and space complexity due to the ever-expanding logs that each client must maintain in order to ensure robust collaboration in eventually consistent environments. This paper proposes an awareness Conflict-Free Replicated Data Type (CRDT) library that provides the tooling to support an eventually consistent, decentralized, and robust multi-user collaborative environment. Our library is tuned for rapid iterative updates that communicate fine-grained user actions across a network of collaborators. Our approach holds memory constant for subsequent writes to an existing key on a shared resource and completely prunes stale data from shared documents. These features allow us to keep the CRDT's memory footprint small, making it a feasible solution for memory constrained applications. Results show that our CRDT implementation is comparable to or exceeds the performance of similar data structures in high-frequency read/write scenarios.

ACKNOWLEDGMENTS

I want to thank my parents and brother for their encouragement, positivity, and support. I'm fortunate to have you all by my side.

I'm very grateful to my grandfather, John, who has always been my intellectual inspiration. So much of my work is derived from our morning coffee talks in the redwood forests of Felton.

Thanks to my grandparents, Karl and Marliss, who helped support my academic endeavors throughout my undergraduate and graduate education.

This project would not have been possible without the council of Kevin Jahns and Bartosz Sypytkowski, who introduced me to CRDTs. Working with both of them on the Y-CRDT project has been a pleasure.

I'd also like to thank my advisor Maria Pantoja, who helped me take ShelfAware from inspiration to publication.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF LISTINGS

Chapter 1

INTRODUCTION

It is challenging to build distributed collaborative software that requires rapid synchronization of replicated states with the standard data structures of modern programming languages. While developers may be able to write custom synchronization code for smaller applications with fewer instances of inter-dependant shared state, they will likely run into unexpected issues with more significant projects where unexpected dependencies and network inconsistencies may lead to synchronization failures. In large applications, segments of state may sync perfectly well in isolation, but the emergent properties of their coexistence often leads to unexpected bugs. This results in a codebase that is difficult to maintain, as new features break the application with additional merge rules that the developers did not expect.

Conflict-Free Replicated Data Types (CRDT) are data structures that ensure the correct merging of states in a multi-user environment. CRDT updates are associative, commutative, and idempotent, which ensures that regardless of the order updates are received and the number of times the updates are applied for two distinct nodes, each node will arrive at the same final state [61]. With these properties, CRDTs can supply the building blocks of multi-user systems. Advantageously, apps developed with CRDTs can work completely offline while preserving the collaborative guarantees of the system. Therefore, developers can build software as if it was intended for a single user and yet confidently ship complex collaborative features using the same data structures. Since the order of the updates does not matter, users can work concurrently offline and automatically merge their state when they reconnect to the network. Therefore, CRDTs allow each user the complete freedom to act individually

while providing collaborative capabilities for the group. Furthermore, this property allows bundled updates to be communicated transitively. While the exact number of message passes to reach consensus differs by implementation, a network of CRDTs often only need to send a total of N to $2N$ messages to reach consensus, where N is the number of clients in the network [2, 55].

Collaboration poses different challenges to the application user, primarily related to awareness. Awareness is the user's passive understanding of updates occurring in the application environment. When multiple users work together in the same environment, each user must implicitly understand the intention of the other to perform effectively together in real-time. This understanding is necessary for peers to make compatible changes and maintain their focus in constantly evolving virtual environments. By predicting the subsequent actions of their peers using the prior knowledge provided by awareness, users can plan out their work in ways to complement the work of their peers, making awareness an integral part of the collaboration. For example, collaborators working concurrently on the same written report will often work on different sections to avoid conflicting with the work of their peers. Awareness information about the cursor location of coworkers allows a collaborator to passively understand what is being handled by others and strategically contribute to unclaimed sections. A key advantage of awareness is that users communicate and receive it passively. The awareness sender automatically generates information as they interact with the application, and the receiving user recognizes the information through non-intrusive user interface indicators. It provides a low-effort framework to communicate a shared mindset [51, 47, 56].

1.1 Problem

Applying general-purpose CRDTs to awareness data leads to systems with high time and space complexity, which can inhibit the functionality of applications. User awareness information consists of real-time high-frequency updates between multiple peers. This update environment means data structures modeling awareness state must update quickly and maintain a small memory footprint. However, to support merging updates from any time in a shared document's history, CRDTs often store all past changes [30, 55]. A document's history or lineage is the ordered data structures that describe the evolution of the document from its initial state to its current state. In order to account for all adaptations, deleted items are marked with a "tombstone" but not erased from the lineage. For large documents with long lineages, storing all of this historical data can take up a large amount of memory, making it an infeasible solution for small devices. Even for devices that can handle the additional memory overhead, storing this information is unnecessary for viewing the latest awareness data. A smaller and simpler data structure can provide these same eventual consistency guarantees.

Performance is another concern when using CRDTs to track awareness information. To integrate new information, CRDTs must sift through an abundance of metadata to find an appropriate location of the new update. While well optimized [34], iterating through any lineage can be a costly proposition compared to the $O(1)$ operation of setting a key in a Map data structure. The difference in performance becomes increasingly apparent with high-frequency updates that are time sensitive since a slight delay can communicate different intentions to peers. Awareness CRDTs should not be the performance bottleneck of the application because this informative data is of secondary importance compared to the user's ability to act on their intentions.

Figure 1.1: CRDT Playground
We designed a simple user interface to visually test our CRDT. Here we see several user cursors in our collaborative CRDT playground.

If the synchronization of awareness information slows an application to an unusable state, these updates become useless.

To our knowledge, only a couple other CRDTs target this specific niche of information distribution [25, 45]. Nevertheless, awareness is a commonly expected feature across many collaborative applications [17, 66, 73]. Therefore we believe that a simplified and efficient CRDT targeted specifically at awareness information could be an effective tool for application developers.

1.2 Contributions

We offer a Conflict-free Replicated Data Type tuned to communicate awareness information with low time and space complexity. Our approach limits update sizes by

passing delta states between clients and uses type-based partial ordering to merge new states rapidly. In simulated awareness-related scenarios, our implementation is competitive with or exceeds other production-grade CRDTs in performance and size. Our contributions include:

- A content agnostic Shelf CRDT implementation.
- An Awareness data structure built on our Shelf CRDT, with memory optimizations for sharing information across peers.
- A delta-state update cycle for the Shelf CRDT that decreases the overall update size for communication between peers.
- A secure update cycle for the Shelf CRDT, ensuring that the CRDT can continue functioning in a byzantine environment.
- A JSON fuzzer, which generates pseudorandom JSON data structures to test the validity of our implementation.
- A set of benchmarks to test performance and memory footprint of CRDTs.

Chapter 2

RELATED WORK

This chapter will cover the foundational contributions upon which we build our work. First, we will discuss the decentralized model of distributed networks. Second, we will cover ordering and consistency in distributed systems. Third, we will discuss how these orderings are implemented in practice using logical clocks. Fourth, we will examine the theoretical models of distributed collaboration systems. Since we base our system on the Conflict-free Replicated Data Type model, our analysis will focus on several different types of CRDTs and their production implementations. Fifth, we will discuss techniques to protect CRDTs against attacks from byzantine nodes in the network. Finally, we will examine the intersection of awareness metadata and CRDTs.

2.1 Peer-to-Peer Software

Peer to peer software is an application architecture where where the clients are responsible for data storage and sharing. In contrast to the classic client-server model, where user devices are only used to interface with a centralized source of truth, peer-to-peer software clients are responsible for data storage and transmission [50].

This comes with a set of significant advantages. Peer-to-peer networks can be more robust than centralized networks, due to redundancies in data storage and transmission pathways. That is, if several data-storing nodes in the network crash, online clients can still transmit and store data on the working nodes in the network. Furthermore, data ownership and management is done locally by each user of the network.

This opens possibilities for users to silo sensitive data on their own devices and share important information directly with a peer instead of going through an algorithmic mediator [3].

However, there are important disadvantages that come with a decentralized peer-to-peer network. For instance, the performance of data transmission within the network is directly dependant on the size and power of the compute nodes that make up the network. If certain regions of users are few in number or lack sufficient compute, they will experience network delays. Additionally, identity and security are hard to enforce in a peer-to-peer system. Since there isn't a central source of truth and data is often shared transitively between nodes, it is difficult to enforce security rules in the network [3].

Our awareness CRDT is equipped to operate in peer-to-peer environments. As long as the network graph is connected, updates will be able to transitively percolate to all nodes, ensuring eventual consistency of state throughout the system.

2.2 Ordering and Consistency

In concurrent computing environments, it is necessary to understand the ordering of events taking place within the system to preserve the intention of the operations being executed. Neglecting this precedence will lead to nondeterminism caused by race conditions: concurrent operations that sporadically execute in different sequential orders. Race conditions can cause programs to behave erratically, crash, or return incorrect results. Based on the needs of the system, there are several consistency models that require different forms of ordering.

2.2.1 Partial Ordering

Partial ordering denotes a relationship r where all elements are not all directly comparable, but can be ordered transitively based on valid relationships to another element in the set:

$$\forall x \exists y.(r(x,y) \vee r(y,x))$$

Three conditions must be satisfied to claim that a relationship r is a partial order [21]:

1. reflexive: Each element is related to itself: $r(x,x)$
2. asymmetric: A relation in one direction prohibits the inverse: $r(x,y) \Rightarrow \neg r(y,x)$
3. transitive: Relationship logic can be chained: $r(x,y) \wedge r(y,z) \Rightarrow r(x,z)$

In distributed systems, events are often partially ordered with the happens-before ($\rightarrow$) relationship. In the case that $A \rightarrow B$, event B will take priority in the instance of conflicting information, because event B made those changes with the foreknowledge of event A. In contrast, when no partial ordering exists between events A and B, we cannot make assumptions about which action takes precedence without further information. With this relaxed framework two consistency models emerge: Causal and Eventual Consistency.

Causal consistency requires that operations be executed in sequential order according to the happens-before relationship. When operations do not have a partial ordering, the operations can be safely executed in parallel. Figure 2.1 illustrates that operations A and B must be executed in sequence, while the branches beginning with C and F could be executed in parallel. The final operation H requires that both E and

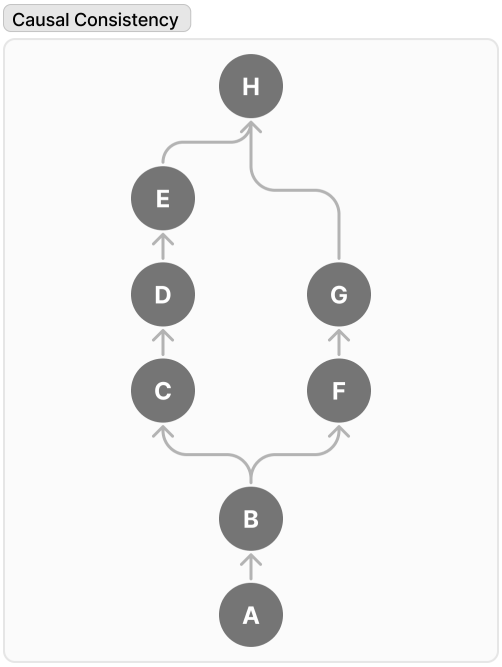


Figure 2.1: Causally Consistent System
Example of event ordering in an causally consistent system

G complete before its execution. Causally consistent systems are frequently used in distributed processing frameworks like Hadoop and Spark. In these frameworks, jobs create a partially ordered lineage graph, where many of the operations can be parallelized across several nodes. Due to the lack of total ordering, causally consistent systems need to keep track of past operations and traverse the ordering to find where an operation should fit in a sequence of jobs.

Eventual consistency is the weakest form of consistency. Its one guarantee is that all nodes that execute the same updates will arrive at the same final state. However, unlike causal consistency, operations could take place in any order. For example, Figure 2.2 shows a communication diagram of a Grow-only Counter (G-Counter) CRDT. G-Counters allow multiple clients to increment a counter value at the same time. Note that client A from B and C apply updates from their peers in differing orders, but arrive at the same final state. A causal system based on logical order would not

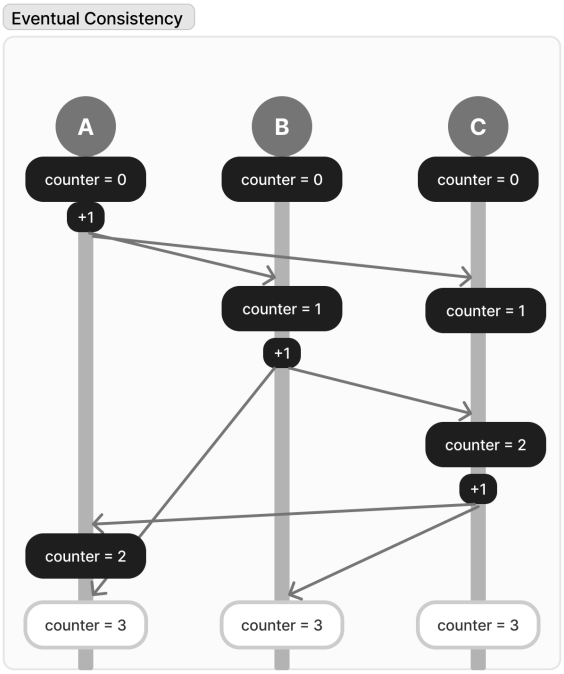


Figure 2.2: Eventually Consistent Example
Communication diagram of three clients updating a distributed Grow-only Counter

allow this to occur. But an eventually consistent system can apply updates in any order, allowing clients to update their state as soon as they receive new information. For software applications, this allows users to make collaborative edits, even if they are disconnected from their peers. The trade-off is that the clients' state varies dramatically in eventually consistent systems. There are $N!$ different ways to apply N updates passed through the system, so consistency cannot be guaranteed in the intermediate states of each client. Our work builds off of the eventually consistent model, proposing an awareness CRDT that sacrifices consistency in exchange for performance and availability. We believe that this is a useful compromise for communicating transient user interactions, but this type of eventually consistent data structure would not provide strict enough requirements for critical data such as financial information.

2.2.2 Total Ordering

Total ordering requires all events in the system to have an order in relation to one another. A system has a total order under relation r if it is partially ordered and $\forall x, y.(r(x, y) \vee r(y, x))$ [21].

Relational databases like PostgresSQL and consensus algorithms like Raft and Paxos keep a total ordering of events that are consistent across every node in the system, but their consistency models differ slightly.

Relational databases subscribe to a strongly consistent model, in which a mutation must be acknowledged by every member of the cluster and written to all nodes before an update is considered complete. This model ensures that all nodes maintain the same state representation at all times, which is important for use cases where accuracy of information is of the highest importance.

Consensus algorithms like Raft and Paxos operate under sequential consistency, which ensures a total ordering of all updates, but allows for certain nodes to have temporarily inconsistent states. Each node may make local changes or assume roles that do not effect the global state on the condition that those decisions can be superseded by operations with a majority consensus. Sequential consistency allows compute nodes to perform actions with majority approval instead of complete approval. This is a desirable attribute in unstable network environments, where a node could be temporarily detached due to a system crash or network failure. Even with such failures, the distributed network can continue to function as long as the majority of nodes are online and functioning.

2.3 Clocks

In distributed networks, time becomes a problematic concept. Even with standardized universal timestamps across the network, compute nodes are likely to have small clock skews, ranging between microseconds to seconds. These small offsets are enough to cause disorder during concurrent operations, since the sequence of operations according to the reported timestamps often does not match the true chronological sequence of operations in the network. To fix this, Leslie Lamport proposed the Logical Clock [42]. A number of logical clocks have been developed, with differing properties. Our CRDT uses logical clocks to create a causal order of updates passed between clients. The following sections outline some of the most common approaches to logical clocks.

2.3.1 Lamport clock

Lamport clocks (Lamport timestamps) are a simple yet powerful method of tracking causality in distributed systems. This type of logical clock is a monotonically increasing integer used for identifying when an update occurred in a partially ordered sequence of updates. When a node receives an update, it compares the Lamport timestamp of that update with the last update applied to its state. If the new update U' has a larger timestamp than the previously applied update U, we can assume that $U \rightarrow U'$. With this information, each node can determine whether updates represent new data compared to its current state, or their information is no longer causally important and should be discarded. Each element in our Shelf data structure is marked with a clock that has a Lamport timestamp component [42].

However, using Lamport timestamps as an identifier can lead to concurrency issues. Two clients can concurrently increment the timestamp simultaneously and produce

two different updates at the same logical time. While these logical updates are locally unique, they share the same logical clock value between the concurrent nodes, which means they must be differentiated using a different heuristic than the clock value. We address this by attaching additional client or value information to the Lamport timestamp to make compound logical clocks that are globally differentiable.

2.3.2 Vector Clock

A vector clock is a collection of Lamport clock representing the known state of each client during an update. For a system with N clients a vector clock is an N element mapping from the identifier of the client to a Lamport lock ($c_i \hookrightarrow \mathbb{N} | c_i \in Clients$). Every client increments the Lamport clock associated with their identifier when they update any shared state, communicating that their latest update is logically caused by the updates that came before. When a remote client successfully receives and applies an update, they merge the vector clock associated with the update with their local state. Vector clocks are merged by prioritizing the maximum Lamport clock associated with a client identifier. Comparing vector clocks allow clients to infer the difference between their current context and the context of the updates issued by their peers. Assume we have two vector clocks V_a and V_b. By comparing the two

clocks element-wise, we can determine the causality of the associated events:

$$\begin{cases} V_a \equiv V_b, & \text{if } \forall k \in keys(V_a) \cup keys(V_b) \rightarrow [V_a[k] \equiv V_b[k]] \\ V_a \rightarrow V_b, & \text{if } keys(V_a) \subseteq keys(V_b) \wedge \forall k \in keys(V_b) \rightarrow [V_a[k] \leq V_b[k]] \\ V_b \rightarrow V_a, & \text{if } keys(V_b) \subseteq keys(V_a) \wedge \forall k \in keys(V_a) \rightarrow [V_b[k] \leq V_a[k]] \\ V_a \parallel V_b, & \text{if } \exists k_1 \in keys(V_a) \cap keys(V_b), k_2 \in keys(V_a) \cap keys(V_b) \rightarrow \\ & [(V_a[k_1] < V_b[k_1] \vee \exists k_3 \rightarrow k_3 \in V_b \wedge k_3 \notin V_a) \\ & \wedge (V_a[k_2] > V_b[k_2] \vee \exists k_3 \rightarrow k_3 \in V_a \wedge k_3 \notin V_b)] \end{cases}$$

Since the logical state of every known client is encapsulated in a vector clock, we can determine that V_a event happened before V_b if V_a and V_b are not equal, V_b has all of the clients in V_a, and all of the Lamport clock in V_b have greater than or equal value to their corresponding element in V_a. When some Lamport clock are greater in value and some are lesser, this indicates that the operations happened concurrently, and each had contextual information that wasn't available to the other. In this case, precedence of operations must be determined using some other mechanism. For our CRDT, we decided against using vector clocks since their size grows linearly with the number of clients. Most awareness information is personal, written by a single user, so conflicting writes to the same location are rare. Furthermore, our CRDT *get* and *set* operations, which do not require the same level of granularity as other data structures that support sequential inserts.

2.3.3 Hybrid Logical Clocks

Hybrid logical clocks (HLC) are a combination of the Lamport clock and Distributed Physical Time. HLCs are monotonic and causal, embodying the traits of a logical

clock with guarantees to remain within a set error range of physical time. With HLCs, a distributed program can effectively monitor happened-before relationships between nodes while maintaining a marginally accurate representation of physical time.

The algorithm has a logical clock and a counter. The logical clock is set to the max of the previous logical clock and the physical time. If the new logical clock is equal to the previous, the counter is incremented, otherwise it is reset. When a message is received, a similar process is performed, while taking the message timestamp into account for the max.

While we did not use HLCs, they could provide useful information for awareness. A lineage-based awareness system using HLCs could keep track of all updates in a history page and display their approximate times to users.

2.3.4 Dot Clock

A dot logical clock is represented by a pair of identifiers: a client identifier with a Lamport clock. Similar to the strategy of a vector clock, pairing the client id with the Lamport clock ensures that the representation is unique and robust against concurrent operations, since only a single client can produce updates with a given id/clock pairing. But unlike a vector clock, the space complexity of Dots remain constant as the number of network clients increase. Additionally, Dots can be efficiently packed into update messages, allowing clients to share only the clock values that differ between their states. Dot clocks are ideal for data structures where low memory overhead and small update sizes are important, which is the criteria for most awareness use cases.

In situations that require causal consistency, dot clocks can be managed by a dot context, which tracks all updates: those that have been applied and others that are waiting for their causal predecessor. For example, if we had a distributed vector of

values where insertions are performed relative to a provided element, any insertion *insert(element: a, before: b)* requires that element b exist before the operation can take place. When a client receives this update, it can check its dot context to determine if the dot clock for b exists, performing the update if it does, and queuing the update for later if not.

The structure of dot contexts are designed to communicate the entire causal relationship of updates with low memory overhead. A dot context is a set of dot clocks paired with a single vector clock. The vector clock represents a lower bound, below which all clients have successfully applied all updates. As clients apply more updates, this bound increases, and any dot clocks in the set that fall below the bound can be pruned. The dot clock set represents a pool of received updates that are missing causal dependency. When new updates with established causal dependencies are received by a client, the client queries the dot clock set for any deferred operations that depend upon the new update. If found, the operation can be retrieved and performed, allowing the client to dequeue its associated dot clock and set the vector clock bound equal to the dot clock's value. This strategy allows for causal systems to function in unstable network environments without passing around redundant states in the vector clocks of updates. Furthermore, in the best-case scenario where updates are delivered in causal order, the dot context has the same space and time complexity as a single vector clock.

We use dot clocks to represent logical time at each node and leaf in our CRDT data structure. dot clocks enable us to distinguish mutations between clients and over logical time, allowing our message passing algorithm to pass small delta-state updates between clients. While a dot context would provide more granular information about the dependencies of concurrent edits, our data structure can ensure convergence without tracking strong causal consistency. To reduce the CRDT's metadata overhead

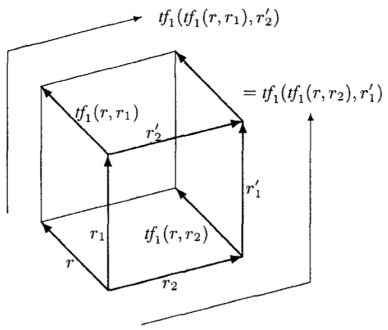


Figure 2.3: Operational Transformation
A central server transforms commands from clients to operate on a shared state [62]

and time complexity, we store individual dot clocks and resolve concurrent conflicts using the ordering between the update and locally stored values.

2.4 Operational Transformation

Operational Transformation (OT) [62] is a conceptual framework that allows multiple users to interact concurrently with an application. In practice, OT systems are fully centralized, and application clients must have an active connection with a server to interact with the application. Text editing software like Google Docs uses OT to enable writers to work together on the same document. The OT server acts as the arbitrator of the application state. When two users update some shared state, the server compares the current state of their applications, and transforms the newer update for the older, applies the changes, and sends back a congruent updated state to both users.

The central algorithm of this technique is the transformation function (tf) which transforms one operation into the context created by another operation. The semi-lattice of Figure 2.3 shows how different operations by clients need to be coordinated

and transformed by a central server to achieve consistency. The closest lower vertex of the cube represents the initial point before three concurrent updates (r, r_1, r_2) are applied. The furthest upper vertex represents the convergent point, where all three updates are merged into a common representation. Each edge of the cube represents the change to the state made by an update. The figure visually represents the symmetry of updates, showing that there are six distinct total orders in which the updates can be applied. The transform function is used to change the origin point of subsequent concurrent updates, but the direction of each update remains parallel to the original, showing a preservation of intention and effect. The exact path that the OT system chooses is irrelevant in practice, under the condition that all paths are proven to converge at the same final state.

This approach is often applied to collaborative text documents. String editing operations based on character indices need to be transformed based on concurrent operations before being applied to a common document. If two users requested the following updates: $r_1 = \text{Add}(\text{“}x\text{”}, 1)$ and $r_2 = \text{Delete}(2)$, then the second update must change with respect to the first. By inserting a character before the deletion, the original deletion index is one less than expected. The resulting transformation function is $tf(r_1, r_2) = (\text{Add}(\text{“}x\text{”}, 1), \text{Delete}(3))$.

The theory behind Operational Transformation is foundational for our work. Our data structure must also have a convergent semilattice of states regardless of the order that the updates are applied. However, our system is decentralized in nature. There is no need for a central server, because we know that upon complete traversal of the semilattice, regardless of the order, all clients will find the same final state.

2.5 Replicated Abstract Data Types

Replicated Abstract Data Types (RADT) are a category of data structure similar to CRDTs. They optimistically change local data and broadcast those mutations as updates throughout the network. RADTs apply updates using local and remote algorithms. The former usually mirrors the classical computer science algorithm and the latter relies on partially ordered operations with higher complexity. RADTs are eventually consistent, but rely on causal dependencies when managing sequences of elements like arrays. Three common RADTs explored by Roh et. al. are Replicated Fixed-size Array (RFA), Replicated Hash Table (RHT) and Replicate Growable Array (RGA) [63]

The most commonly referenced RADT is the Replicated Growable Array. An RGA represents a sequence of elements that can be edited with insert, update and remove operations. Internally, the data structure is a linked list of nodes which consist of a pointer to a hash table, the stored value, and the next item in a linked list. The hash table is used for O(1) retrieval of a given node using an identifier vector. The linked list defines the order of the nodes and allows for easy iteration. The insertion operation specifies the location of the new item by referencing the element that it will be placed after. Items that are concurrently inserted are ordered by precedence, with the highest precedence item being closest to the defined left anchor. Unlike many CRDTs, RGAs have the ability to prune deleted values, which can save memory. The primary advantages of RGA are O(1) operation times, dynamic sizing, and prune-ability [63].

Similarly our CRDT can prune deleted states. Unlike RGAs, our data structures are not equipped for sequences of elements. Arrays are treated as atomic units, where concurrent set operations overwrite each other based on a developer-provided partial

ordering for their specific sequence data type. This tradeoff allows us to simplify the CRDT to resolve all updates under a framework of eventual consistency instead of queueing dependent updates in a causal framework. The result is an overall reduction in CRDT metadata.

2.6 Conflict Free Replicated Data Types (CRDT)

Conflict-free Replicated Data Types are decentralized abstract data types that share information between collaborators by passing update messages over a network connection. They serve as composeable building blocks for collaborative programs, which abstracts away the need to manually reconcile states across peers. Simultaneously, CRDTs work with an optimistic update framework, which means any client can change shared state locally without needing to confirm with remote peers. As a result, CRDTs allow users to work as unconstrained individuals, even when collaborating on shared documents [61, 67].

To enable this flexibility, all CRDTs must uphold a few core traits. The most important requirement is that a CRDT must be eventually consistent, that is, all nodes that receive the same update messages from its peers must converge to the same state. This requirement is important because ultimately CRDTs represent a single true shared state, though clients may arrive at that final convergence on many different paths. The flexibility in how this state is reached allows CRDTs to operate in a way that is most convenient for the user, while ensuring all users arrive at the same conclusions when they ultimately synchronize. Additionally, the application of updates in the synchronization process must uphold three traits [67]:

1. commutativity: updates can be applied in any order, producing the same final state.

2. associativity: updates can be combined in any order, producing the same final state.

3. idempotency: updates applied several times to a CRDT produce the same final state.

An additional advantage is that none of these requirements specify a communication method. As long as the updates eventually reach all clients, it does not matter the means by which they arrive. Some projects like Ditto [16] leverage a chain of many communication methods to communicate updates between CRDT nodes.

CRDTs are actively used for a number of enterprises. Apple uses CRDTs to synchronize application state between devices which share iCloud accounts [17]. Text editors like Atom use CRDTs to enable multiple developers to work on the same coding document [66]. Databases like Riak [6], DynamoDB [15], and Antidote DB [58] use CRDTs to enable high availability of rapidly-updating shared states across partitions.

Our Awareness CRDT allows for the rapid synchronization of client metadata while keeping the time and space complexity of the data structure low. The synchronization process upholds communication ambivalence and can piggyback off of the preexisting network of applications.

2.6.1 Centralized CRDTs

In contrast to Operational Transformation, CRDTs can exchange messages directly between clients without needing a centralized server to resolve conflicts. However, for performance and architectural ease, it is sometimes preferable to use CRDTs in a client-server model. This is often the case when developers use a CRDT for an

auxiliary feature atop a centralized application. For example, the UI design company Figma implemented a collaborative editing mode in their design software using a CRDT [77]. For performance and consistency, the server is the arbiter of the design document's state, which is represented as a tree of UI elements. Each user interface element has a series of attributes that can be collaboratively update based on the Last Write Wins (LWW) heuristic. LWW states that the latest updater in a series of changes overrides the work of the previous editors. When a user edits the document, the server use the Last-Write-Wins heuristic to compute the updated state and distribute the update among the clients. Since all data is orchestrated by a single server, the collaborative CRDT does not have to store a lineage of past edits, which reduces its memory footprint and increases its throughput. This is a key advantage over Operational Transformation, which must keep track of past edits in order to integrate delayed updates.

2.6.2 State-based CRDTs

State-based CRDTs (CvRDT) are decentralized data types that update with peers by sharing information instead of transmitting operations performed on the data type [67]. An example of CvRDTs with collaborative text editing would be clients sharing their document's text state with each other instead of sending individual character insertion operations. CvRDTs come in many forms, but they often keep track of the age of their attributes with Lamport timestamps, which are monotonically increasing counters that increment due to a change in the data structure [42]. When the program updates an attribute, the CvRDT increments the Lamport timestamp and disseminates the mutated state among the peer network. The receiving peer can safely overwrite the value if the new state has a timestamp of greater value than the one is currently has. This makes updates rather quick, since the CvRDT

simply writes the updated values instead of calculating the resulting data structure by applying a sequence of operations. For example, if the CvRDT receives an update with a timestamp of 5, it can safely ignore delayed messages with updates 4, 3, and 2, since update 5 succeeds the others and therefore contains the state of previous messages. State-based CRDTs are ideal when the total amount of data is small, the updates are frequent and the operations necessary to calculate the state are expensive. Conversely, large memory footprints require CvRDTs to send large updates over the network, slowing down the receiving clients. Delta State CRDTs have aimed to improve network performance by sending a smaller representation of the state, instead of shipping the entirety of the CRDT to each client [2]. Each Delta CvRDT instance uses prior knowledge of what other clients already have in their local state to only send novel information in their delta updates. Our work leverages the delta-state technique to reduce the size of data that we send between clients, in light of time-sensitivity of our Awareness communications.

2.6.3 Operation-based CRDTs

Operational CRDTs (CmRDT) communicate commutative and associative commands between peers that operate on the CRDT to reach a convergent state. When a program updates the CRDT locally, it does so via some function which gets, sets, adds or removes data in some fashion. To apply a similar change to its peers, Operation-based CRDTs share the operation that will bring the peer's current state to the desired final state, along with the contextual information necessary to carry out the mutation. These operations must be commutative and associative, since they could arrive in any order due to network delays. In order to offer this CmRDT have to specify their commands such that they can work on clients with completely different versions of the CRDT state. For example, CRDTs often define sequential data like strings

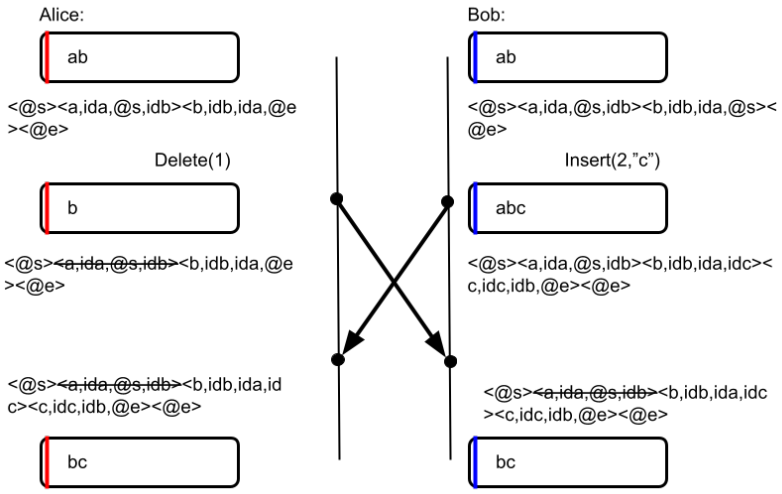


Figure 2.4: Sequence-based Conflict-Free Replicated Data Type for text editing.

as linked lists. Inserting a character into a word by index would be highly context specific and lead to divergent results, since each client could have strings of different length with different characters when they receive the insert operation. Instead, Operation-based CRDTs specify the neighboring characters of the insert operation. Even if other operations move around the expected index, the insert operation will be deterministic and correct, since it is relative to two neighboring nodes. This allows Operation-based CRDTs to iteratively update and arrive at the same final state [57]. Operation-based CRDTs are highly effective at updating large data structures when updates are infrequent and the computational complexity of running the commands is low. Since the update payload describes an operation, it is usually very small and can be easily passed to peers. However, CmRDTs often must apply all commands in order to reach a convergent state, which could lead to a high computational load.

Table 2.1: CRDT Libraries Summary

CRDT	Advantages	Disadvantages
WOOT	eventually consistent text editing, causally consistent updates	large tombstones, must maintain an unbounded queue of updates
Yjs	JSON data types, compact tombstones, undo/redo support, rich text editor support	expensive list traversal, requires tombstones
Yjs Awareness	low memory overhead, efficient get/set operations	redundant data sent in updates
Automerge	general purpose, efficient bloom filter update comparison, immutable documents, functional update system	large lineage memory overhead, slower operations than other CRDTs [26]
Peritext	preserves authors intent with rich text	doesn't support composable data structures
Shelf CRDT	pruneable, simple, performant	can't represent collaborative sequences

2.6.4 Text Editing CRDTs

CRDTs are particularly useful for building collaborative word editor applications. Our work is designed to be used in conjunction with text editing CRDTs to provide information like cursor position and highlight ranges. Due to this complementary relationship, we will review some of the industry standard CRDTs used to create text editing software. Table 2.1 provides a summary of the advantages and disadvantage of commonly used CRDT frameworks.

2.6.4.1 WOOT

WOOT is a CRDT text editing algorithm that ensures distributed convergence between clients with a single round of message passing [57]. The data structure uses individual characters as the atomic unit of operation, which is represented by a 5 tuple of *(clock, character, tombstone, prev, next)*. The *clock* is a Dot Logical Clock uniquely identifying the character in the text sequence. The *character* holds the value

of the character to be displayed to the user, while the *tombstone* is a boolean marker that denotes when a character has been deleted. The *prev* and *next* fields determine the relative location of the character in a word by storing pointers to its neighboring characters. The WOOT CRDT has two operations: insert and delete. Deleted characters are marked by tombstones and not shown to users. Insertion operations are identified based on the characters adjacent to the character being added. In the case that a referenced character for inserts don't exist yet at the source, the algorithm keeps a pool of non-executable commands and checks the pool after each new update. Inserts have the additional possible technicality of unexpected characters existing between the referenced neighboring characters. Given two character nodes C_a and C_b, insertion conflicts are resolved by the relative ordering:

$$C_a > C_b \iff C_a.\text{prev.index} > C_b..\text{prev.index} \vee C_a.\text{next.index} > C_b..\text{next.index} \vee char(C_a) > char(C_b) \quad (2.1)$$

WOOT stands as a foundational reference design for text-editing CRDT due to its preservation of user intention across clients and efficient implementation. However, it is not optimized for awareness information associated with text documents, such as cursor position. Our work would be an effective compliment to text editing CRDTs like WOOT, handling the responsibility of sharing user metadata between peers.

2.6.4.2 Yjs

Yjs is an operational CRDT written in JavaScript. Built on the YATA theoretical framework [54, 55], it is highly optimized for concurrent drawing and text-editing

applications. The system is able to reduce its memory footprint significantly by compacting adjacent CRDTs into a shared representation. Instead of storing every character as a node in a linked list, Yjs pools adjacent characters written by the same author into a single node. Deleted characters are also compacted into grouped tombstones and with their contents erased. Furthermore, the library encodes operations to binary state vectors before communicating updates to peers, which decreases the overall network load [24]. For common editing benchmarks, Yjs is considered the best performance CRDT on the web [26]. Our work aims to provide a faster, more secure awareness API that integrates with the Yrs ecosystem, which is the successor to Yjs. The current Yjs awareness CRDT is a classic CvRDT, where the entire state is shared on each subsequent update. This strategy can be costly for small and frequent state updates by clients. While the Yjs updates can be encrypted for security, it is not provided by the library. Our work uses delta state updates to reduce the size of updates passed between CRDT instances and proposes a framework for securing Shelf CRDTs against bad actors intent on corrupting the CRDTs contents.

2.6.4.3 Automerge

Automerge is a tree-based operational CRDT for local-first collaborative applications. Local-first software stores data on client devices, allowing users to own their data. Local-first applications with collaboration necessitate that the lone user has the same capabilities as one actively connected to the group. Automerge's structure allows for rapid search and deletion throughout the data structure, leading to a robust CRDT implementation. As the user interacts with the CRDT, it stores a list of operations, which it compiles into a batch called a change. This change is shared between clients in order to synchronize state. However, the sending CRDT does not know which updates its peer needs in order to reach consensus, so it sends out a hash of its

internal state. The recipient compares this hash against its own state using a bloom filter and determines which attributes it is missing. By passing this information back to the sender, the sender can encode a change set with the missing state and update the recipient. For text editing, Automerge stores strings as linked lists of multi-valued character registers. This guarantees that concurrent updates are both stored, but has strange duplication side effects which can be counter to the intentions of the document's authors. An additional issue is its ever-growing data structures, which leads to large document sizes in memory. For our Awareness API, we use a pruneable Shelf CRDT. Pruneable data structures can eliminate stale data completely without storing tombstones. This decouples the memory footprint of the CRDT from the size of its history, allowing it to undergo a large amount of rapid updates without growing in size [36].

2.6.4.4 Peritext

Peritext is a modern text editing CRDT that draws its inspiration from Yjs and Automerge. Its primary goal is to preserve the author's intention during collaboration, covering specific edge cases where other CRDTs fall short. Specifically, Peritext allows authors to collaborate in rich text, which requires the disambiguation of text styling applied concurrently by multiple users. In previous works [36, 59], overlapping styles often overwrote each other or led to unexpected effects at their intersection. Peritext is able to handle these cases due to its novel meta data approach to text document representation. It stores the document as a linked list of characters, where each character node stores markers to indicate the start and end of a style. The node stores beginning and ending marks that dictate the style of a range in two sets: marks the precede the letter and those that succeed the character. This dictates whether future inserts adjacent to marked characters adopt the styling and solves the

issue of the user accidentally deleting hidden style tags. This strategy of tag adhesion makes the effects of concurrent cooperative editing better reflect the intention of each individual author [43].

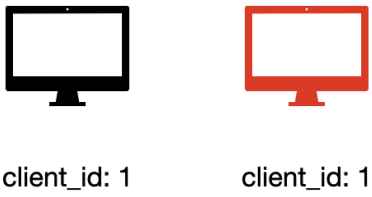
2.6.5 CRDT Security

The relaxed requirements of CRDTs and their decentralized nature makes security constraints hard to enforce. Clients must rely on each other to pass information through the network, making man-in-the-middle attacks a prevalent concern. Unlike client/server models, there is no central authority in the network to enforce authentication and trust standards across the network. Whatever enforcement procedures are put into place need to work in low-trust environments where malicious actors can read and write to the CRDT. The goal of CRDT security is to ensure that bad actors are unable to degrade the eventual consistency guarantees of the CRDT.

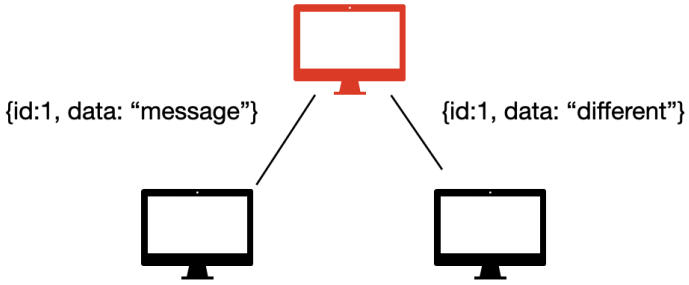
2.6.6 CRDT Exploits

Due to their abstract nature, CRDTs can uphold their properties in systems with ill-defined security roles. However, attackers can invalidate the traits of a CRDT by maliciously editing the information communicated during updates. The following two exploits can be leveraged to destroy the eventual consistency guarantee of a CRDT:

- Client Spoofing: A bad actor uses an identifier of another client to publish malicious messages.
- Duplicate Update Identification: A bad actor communicates two different updates under the same logical clock identifier, causing the CRDT to become irreparably inconsistent across clients.



(a) Client spoofing



(b) Duplicate update identification

Figure 2.5: Examples of CRDT exploits

The goal of CRDT security is to prevent actions that corrupt or hinder the functioning of the distributed update system. Application security is an additional and necessary concern when using CRDTs in production. However, it differs between cases and depends on the composition of the shared state. We will primarily focus on ensuring that malicious actors cannot violate the traits of a correctly functioning CRDT.

2.6.6.1 Operational Hash Graphs

Hash Graphs have been proposed as a framework to secure operation-based CRDTs. In this system, every local CRDT instance has a hash graph of all updates received. This directed acyclic graph allows nodes to compare known updates and establish a

causality relationship between changes. This causality is communicated via a predecessor hash, which encapsulates the identifiers of the updates that immediately causally precede a change. This system protects user identity by using the hash itself as an identifier. Bad actors cannot replicate a user's hash because no identifying information is communicated. The predecessor hash system also ensures that CRDT guarantees are maintained, since the hash is directly tied to the contents of the update being communicated. Our work leverages hashes in a similar way, but attaches them to parts of the CRDT state instead of operations. [32]

2.6.6.2 Merkle Search Trees

A Merkle Search Tree (MST) is a combination of BTrees and Merkle Trees. It is similar to a BTree in its data structure, storing B elements in each node and upon overflow splitting its contents into further leaf nodes. Each of these nodes is deterministically ordered and will form the same tree regardless of the order of insertions and deletions. Hashing the keys recursively up from the leaves forms a Merkle Tree, where root hashes can be compared in O(1) to check if two trees have the same elements. MSTs can be effectively compared via an anti-entropy update process by which updates percolate throughout the network at log(n) speed and require a number of passes equal to the tree depth to come to consensus. Two comparing clients can walk the differing hashes in the tree down from the root in order to find the leaf elements that differ. This is much more efficient than comparing the entire tree for large data structures and preserves bandwidth by only sharing hashes, but it suffers the cost of recursively rehashing upon every change. Our work has taken inspiration from MSTs, using hash comparisons in order to ensure that updates are consistently applied at each remote client. [4]

2.6.6.3 Secure Universal Composability Framework

The Universal Composability Framework for CRDT security defines two approaches for implementing secure data structures: (1) encrypt the CRDT updates without changing the underlying algorithm, or (2) build encryption into the underlying merging algorithm, using homomorphic encryption to combine states without decrypting the update's contents [5].

The first solution works best for CRDTs that require only an equality trait between stored values. Registers and sets can store encrypted values using deterministic hashing without any knowledge of the decrypted values. The second solution is required for CRDTs that require mutations to the values that are stored within. Counters are a good example, since the increment operation must read the previous value and write back the incremented one. One solution is a homomorphic encryption scheme that allows the server to perform additions without prior knowledge of the actual value [1].

Our system verifies that the relationship between the clocks and values in our CRDT remain intact. This ensures that byzantine nodes cannot pass off corrupted values as identical to the originally set value during transitive steps in the update process. While ShelfAware does not provide encryption as a library feature, all of its messages can be encrypted before transmission with commonly-used cryptography libraries [72]

2.6.7 CRDT Awareness

Collaborative software, like text editors, sync a shared application state between users. To improve users' collaborative experience, the application can share and display user activity information like cursor locations with different identifying colors and names.

This kind of information is called *awareness*. Its primary purpose is to give users hints about others' intentions and where on the document they are currently working on or intending to work. Awareness CRDTs require a different set of trade-offs than standard text-editing CRDTs. Instead of dynamic sequences of text, awareness state can often be represented by get and set operations in a user state map. For this reason, awareness CRDTs often sacrifice complexity for memory efficiency and performance. For example, The Yjs Awareness CRDT operates like a LWW-Register [67], while the Yjs text editing system uses the more expressive YATA framework. Our implementation enables more structured data types, allowing for a recursive tree of values instead of tracking individual client changes in a single register.

2.6.8 Shelf CRDT

The Shelf CRDT is a compact data structure that enables simple conflict resolution between clients. A Shelf supports all JSON data types; however, it treats sequences such as arrays and strings as atomic elements. This restriction makes Shelf CRDT unsuitable to track the shared sequential edits that are commonplace in collaborative text editors. Nevertheless, many use-cases do not need tracked sequences, like tracking user metadata in an application. This makes Shelf CRDT a good fit for sharing awareness information in applications [45].

The original Shelf CRDT was famously implemented in under 90 lines of JavaScript by Greg Little [27]. We built upon this algorithm in several ways. ShelfAware updates peers by passing compressed delta-states, which limits the amount of information that has to be communicated between clients. We also provide an interface for the client to update the Shelf directly. This allows us to update portions of the data

structure instead of recursively wrapping the entire state on every update. Finally, our implementation has a customizable and more efficient type ordering system.

This paper's *main contribution* is to provide an open-source CRDT library that can handle a high volume of clients and efficiently merge updates. To prove our approach's validity, we compare the performance and correctness of our system to similar open-source libraries. The ShelfAware code and the accompanying experiments can be found online for replication [1].

[1] https://github.com/Waidhoferj/thesis

Chapter 3

METHODS

In this thesis, we present ShelfAware, an Awareness CRDT library for handling high-frequency updates between clients. In order for peers to collaborate effectively in a shared environment, each individual must be able to passively communicate their intentions to their coworkers. This data is often small, time-sensitive, frequently updated, and of secondary importance to the actual application's business logic. To develop a specialized CRDT for such a use case, we set the following system principles:

- Small Updates: Updates should be small so that their network and local processing impact remains low.
- Fast Operations: Get, set, and merge operations should have low time complexity. This allows the CRDT to keep up with rapid updates.
- Secure: ShelfAware should continue to function even in the presence of malicious nodes. Security ensures that user intention is correctly represented.
- Convergent: ShelfAware should converge and uphold the basic principles of CRDTs. Convergence ensures that all clients move towards a common understanding as they gain more information.
- Composeable: ShelfAware should be an effective companion CRDT for other collaborative systems. Awareness is coupled to other application states, so ShelfAware must be able to work alongside preexisting state management tools.

- Flexible: The Shelf implementation should support common JSON data types out of the box, and provide an extensible interface for user-defined types. Flexibility makes the system adaptable to the needs of the application developer.

This chapter will examine the conceptual framework offered by ShelfAware. This framework reveals the key advantages ShelfAware offers, and the associated limitations caused by specialization. We will begin with an overview of ShelfAware's features. Then the structure of the foundational ShelfAware Shelf CRDT will be discussed. Next, we will describe the operations that can be performed on the CRDT. After, we will review security and how ShelfAware can protect against CRDT-related exploits. Finally, we will critique some of the limitations and trade-offs associated with our system model.

3.1 Overview

ShelfAware is a library that provides a collaborative Shelf CRDT interface for storing highly granular information that is frequently updated. The ShelfAware Shelf CRDT allows users to perform optimistic local updates independently while quickly sharing this information through a network of peers. In addition, the system is eventually consistent, ensuring that all clients arrive at the same final state after disseminating all updates.

Beyond the core guarantees of a CRDT, the library offers features that keep memory usage low. For example, ShelfAware Shelves are pruneable, meaning stale data can be removed entirely from the CRDT. Other CRDT implementations with causally-related updates require preserving deleted values for the CRDT to function. Pruneability results in the ShelfAware's metadata overhead growing with the size of the stored state, not the total number of updates. This trait allows the system to keep

a constant memory footprint during in-place updates. A further memory optimization exists in ShelfAware's update passing system. ShelfAware is a Delta CvRDT that only passes novel information to its peers, reducing the amount of excess data communicated through the network.

Additionally, ShelfAware provides a secure update and merging system that prevent bad actors from violating the convergence guarantees of the Shelf CRDT. The secure interface allows ShelfAware data structures to operate robustly in environments with byzantine nodes that seek to exploit the logical clock system to spread corrupted data.

Finally, ShelfAware is a system built on traits, allowing it to be flexibly applied to various data types and use cases. While we designed this system with awareness in mind, the Shelf CRDT can be used effectively as a low-overhead collaborative data store in myriad contexts.

3.2 Structure

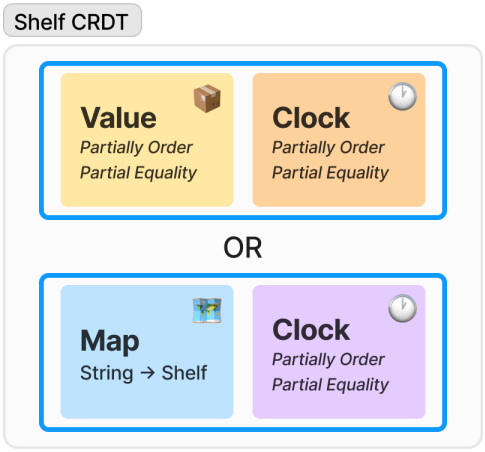


Figure 3.1: ShelfAware Shelf CRDT Structure
Both of the clocks and the contained value are generic

The Shelf CRDT provided by ShelfAware is a structure with two parts: a contained value and a clock. The value of the Shelf can either be a mapping of keys to Shelves or some atomic value with an intrinsic partial ordering. The clock is a partially ordered, monotonically increasing logical counter. The Shelf remains agnostic to the specific implementation of the clock and the contained value, providing developers with extensibility. For example, developers can create partially ordered values to store on the Shelf without changing the system's structure. Likewise, developers can specify custom partially ordered clocks for specialized use cases.

While this composability is helpful for developers working with custom data types, it does provide an additional layer of complexity. For this reason, we supply default value types and clocks that apply to typical use cases. Application programmers can begin developing with these standard components and implement their custom logic for specific use cases if needed.

3.2.1 Provided Values

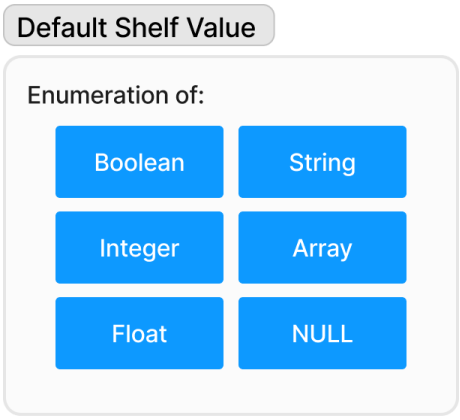


Figure 3.2: Default Shelf Value Structure

The default implementation of the ShelfAware Shelf wraps a Value type: an enumeration of JSON-like data types (Figure 3.2). We chose to make these types the default

ShelfAware experience because it promotes compatibility with our JavaScript library bindings and supports conversion into the commonly used JSON file format.

Like all Shelf contents, the Value type is partially ordered, internal to each type, and between the enumerated types. The ordering between types is based on the type's complexity. The cross-type precedence is Array, String, Integer, Float, Boolean, Null from highest to lowest. The partial ordering internal to elements of the same type uses the default ordering of that type.

3.2.2 Provided Clocks

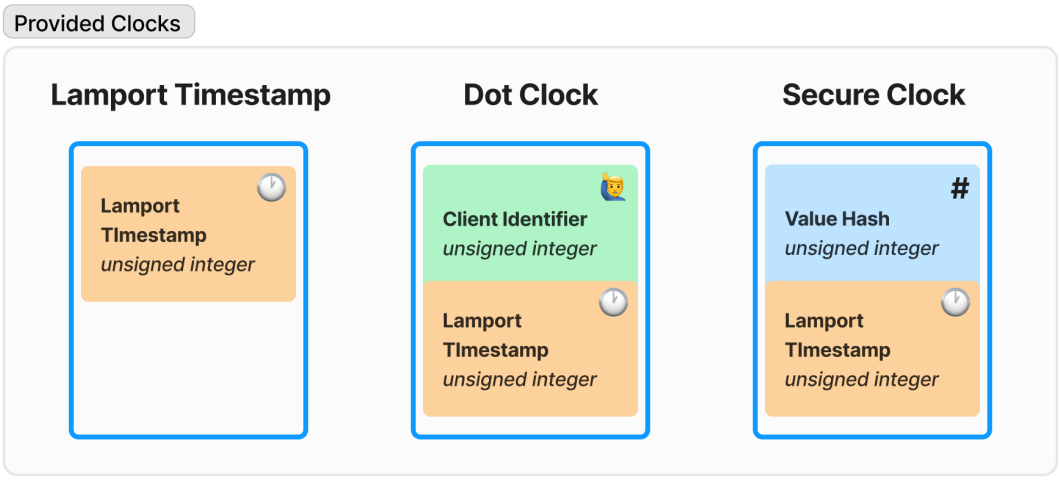


Figure 3.3: Structure of ShelfAware's three provided clocks

The Lamport Timestamp is a monotonic logical clock that increments when a portion of a Shelf receives an update. Lamport Timestamps are the smallest provided clock data structure, requiring only a single integer comparison to deliver a partial order. However, an updating Shelf CRDT cannot differentiate between Lamport Timestamps set by the same user and two timestamps set concurrently by different users. This can cause the incorrect pruning of novel information from delta updates sent between clients. For this reason, we recommend using Lamport Timestamps

as the clock associated with Shelves that are scoped to a single user, or on Shelf Maps. The merging algorithm compares Shelf Maps recursively when clock values equal, accurately combining concurrent updates. Single-user Shelves are suitable for most awareness applications, authorizing users to write only to their scoped state. This scoped mutability allows us to represent an awareness client using Lamport Timestamps.

The dot clock is a tuple clock: $(client\_id, lamport\_timestamp)$, which can effectively distinguish concurrent updates between clients. This clock is an effective tool for Shelves where clients have read-write access to a shared portion of state. While the memory footprint of the clock is minimal compared to their associated Shelf Value, dot clocks are twice as large as the equivalent Lamport Timestamp. In instances where a Shelf has a single writer or infrequent updates, Lamport Timestamps would reduce the metadata overhead.

The Secure Clock prevents malicious clients from spoofing client identifiers or passing different content with the same logical timestamp. The data structure is a tuple clock $(hash, lamport\_timestamp)$, where the *hash* is derived from the Shelf Value and the associated Lamport timestamp. The Secure Clock can effectively replace the dot clock for tracking concurrent changes to shared mutable Shelves. However, the Secure Clock incurs the additional overhead of hashing the associated Shelf Value during initialization and the merging process.

3.3 Local Operations

The Shelf CRDT has two local operations: *get* and *set*. The *get* operation fetches a reference to the value stored inside the Shelf. The *set* operation (Algorithm 3.1) stores a new value in the Shelf and increments the Shelf's clock. The algorithm to

increment a clock is specific to each clock data type, but in all cases it monotonically increases the value of the internal logical clock. An incremented clock indicates an update to Shelf Value so that other clients can request the new information during delta updates.

Algorithm 3.1: SET updates the internal value of a Shelf

```
function SET(shelf, value)
    shelf.value ← value
    shelf.clock ← INCREMENT(shelf.clock)
     return shelf
end function
```

3.4 Communicating Delta Updates

ShelfAware uses a two-step delta update process for syncing data between clients.

First, the update requestor sends out a state vector, which is a skeleton of the recursive Shelf Map structure containing only the clocks of each Shelf. The state vector captures the logical time of all Shelves in the system, allowing other clients to determine which elements the update requestor needs. Passing around the clocks without the associated value reduces the size of update messages, keeping the network overhead of the Shelf update process low.

Next, the update sender compares the logical time of its Shelf CRDT structure with that of the received state vector. Any Shelf structures with a clock value that equals or happens before the corresponding state vector clock holds data that has already been seen by the update requestor. We can safely prune these stale values from the delta update without the requestor losing any information. This technique reduces the overall size of the update. For collaborative environments with localized, high-frequency updates, the delta state will consist of a small subset of the Shelves within

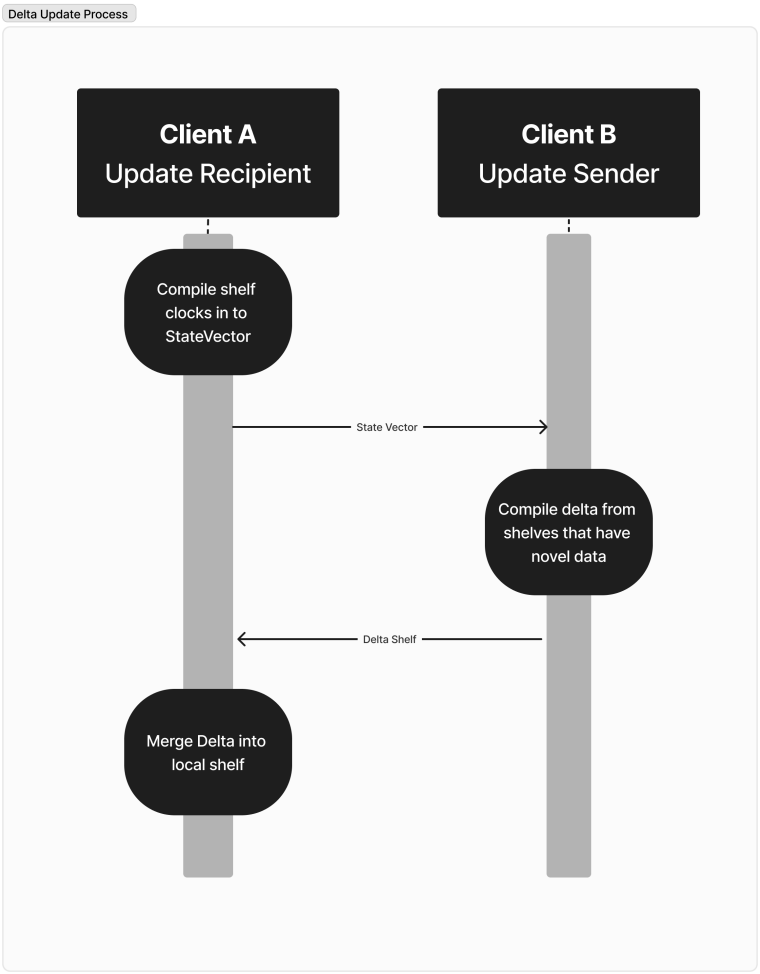


Figure 3.4: Sequence diagram of the delta update process

the shared state, drastically reducing the data quantity clients have to serialize and share with their peers. We compile any data deemed novel by this criteria into a delta update. Delta updates are Shelf CRDTs that hold some subset of the update sender's Shelf state. When the update sender responds with the Delta Shelf, the recipient merges the novel delta elements with their local Shelf state.

3.5 Merging Process

Generally, Shelf CRDTs resolve conflicts between updates and their current state by first comparing clocks and then breaking any ties with an absolute type ordering [45]. As shown in Algorithm 3.2, the data structure resolves conflicts by applying a series of ordering rules:

1. The item with the highest clock overrides the other.
2. If the items have different types, the item with the highest type order overrides the other.
3. If both items are Shelf Maps (they hold Shelf data structures) and both clocks are equal, recursively merge each child.
4. If the items have the same type, the item with greater comparison value overrides the other.

Fortunately, these guidelines allow the Shelf to remain ambivalent about the data that it contains. As long as the Shelf contents are partially ordered, rules three and four can be abstracted away from the specific structure of the Shelf contents. Similarly, the Shelf Map can operate on the premise that it holds either an opaque value or more Shelf Maps. The merging process allows clients to transitively synchronize updates across the network and arrive at a single convergent state.

3.6 Security

To protect ShelfAware from client spoofing and duplicate update identification attacks, we employ secure clocks, which effectively tie the Shelf's contents to a specific

Algorithm 3.2: Merge algorithm

```
function MERGE(shelf, delta)
    if shelf.clock > delta.clock then return shelf
    else if shelf.clock < delta.clock then return delta
    else if IS_SHELF_MAP(shelf) ⊻ IS_SHELF_MAP(delta) then
        if IS_SHELF_MAP(shelf) then return shelf
        else
     return delta
        end if
    else if IS_SHELF_MAP(shelf) ∧ IS_SHELF_MAP(delta) then
        updated_map ← MERGE_SHELF_MAPS(shelf.map, delta.map)
     return new Shelf(updated_map, shelf.clock)
    else
        if shelf.value > delta.value then return shelf
        else
     return delta
        end if
    end if
end function
function MERGE_SHELF_MAPS(shelf_map, delta_map)
    for key in delta_map do
        if key ∈ shelf_map  then
            shelf_map[key] ← MERGE(shelf_map[key], delta_map[key])
        else
            shelf_map[key] = delta_map[key]
        end if
    end for
     return shelf_map
end function
```

logical time (Algorithm 3.3). The secure clock stores a hash of the Shelf contents and the Lamport timestamp associated with the state. Since secure clocks are partially ordered based on their Lamport timestamp, the algorithms associated with local immutable operations and the delta update cycle do not functionally differ.

However, the *set* and *merge* operations require an additional hashing and validation phase.

Setting the value of the Shelf increments this clock and creates a new hash with the new value clock combination. This step ensures that all mutations are associated with a hash that represents both the value and the logical time of the operation. When the Shelf operates in a safe environment, the hash is sufficient to connect the clock and the associated value. However, byzantine nodes could subvert the set operation and edit the value in the Shelf directly in memory without updating the clock. While we cannot ensure that the memory of the Shelf is not tampered with locally, we can validate new Shelf entries during the remote merging process, ensuring that the corrupt Shelf does not propagate through the network.

The secure merge process has the same algorithmic foundation as the standard merge, with an additional validation consideration applied when outside values are added to the Shelf. When a clock comparison indicates that the remote value should be merged into a local Shelf instance, the local instance hashes the incoming content with its associated clock value. If the resulting hash matches the remote secure clock sent alongside the value, the local instance can integrate it with the assurance that the clock value pair is unique. The local Shelf can discard the update if the provided hash does not match the locally produced hash. Integrating a remote Shelf Map involves this validation process over each key and recursively for each sub-Shelf. The Shelf Map's recursive integration filters out corrupt values while integrating all valid entries.

Since the hashing algorithm is deterministic, all clients will discard corrupt updates, which preserves the eventual consistency of the Shelf CRDT. However, we cannot feasibly prevent bad actors from circumventing our library's constraints locally and editing Shelf Values directly in memory. For this reason, corrupted Shelves could still exist on byzantine nodes that are not reflected in valid nodes across the network. Therefore, the eventual consistency guarantees only apply to the valid Shelves verified by hashing constraints. Pragmatically, this allows valid users to safely function within the collaborative network without incurring corruption from byzantine nodes.

Algorithm 3.3: Secure Shelf Algorithms

```
Hasher
function VERIFY_CLOCK(secure_clock, value)
    clock ← secure_clock.clock
    hash ← secure_clock.hash
    hash_is_valid ← Hasher.hash((clock, value)) == hash
     return hash_is_valid
end function
function PRUNE_CORRUPT_CONTENT(shelf)
    if shelf is a Map then
        shelf_ids ← keys(shelf.map)
        for shelf_id in shelf_ids do
            sub_shelf ← shelf.map.remove(shelf_id);
            valid_shelf ← PRUNE_CORRUPT_CONTENT(sub_shel)
            if sub_shelf is not NULL then shelf.map.insert(shelf_id, sub_shelf)
            end if
        end for
        if shelf is empty then return NULL
        else
     return shelf
        end if
    else
        if VERIFY_CLOCK(shelf.clock, shelf.value) then return shelf
        else
     return NULL
        end if
    end if
end function
```

3.7 Limitations

The ShelfAware library provides a performant, minimalistic, and secure framework for rapid awareness communications. Still, the design choices that lead to these advantages have several limitations.

ShelfAware inherits the primary simplification of Shelf CRDTs: collaborative sequence types are not supported. Users cannot concurrently insert or update lists of elements inside a Shelf. When a concurrent update does occur, the change with the higher lexicographic order will override the other adaptation. More feature-rich CRDTs like Automerge [29] and Yjs [55] support ordered CRDT arrays in the form of linked lists where concurrent updates can be merged based on the neighboring elements that they reference. We forego this feature because awareness updates are often scoped or completely replace the previous value. Adding this feature would require managing a queue of causally related array insertions, adding additional performance and memory overhead to the Shelf data structure. Additionally, most CRDT sequence types require preserving deleted values as tombstones, which would increase the size of ShelfAware during rapid updates. We decided to refrain from implementing this feature to preserve our design principles, allowing developers to use other industry-standard CRDTs to provide this data type.

ShelfAware keeps a minimal memory footprint by overwriting older values with new updates. This means that our Shelf CRDT does not have an easily accessible lineage, like the partially ordered hash graphs provided by Automerge [36]. Consequentially, ShelfAware can show the current state but not how it was achieved. If this is a desired feature, developers can store all outgoing delta updates for the duration of interest, order them based on local timestamps, and merge each delta sequentially to produce the partial Shelf states at each point in history.

Additionally, the Shelf data structure is not robust to denial of service attacks which can prevent users from updating a Shelf Value. For example, bad actors could hijack the local *set* operation to produce an update with the logical clock set at the maximum integer value. Such an attack would prevent all other users from updating the value since the logical clock of the malicious update would always be higher than the new updates.

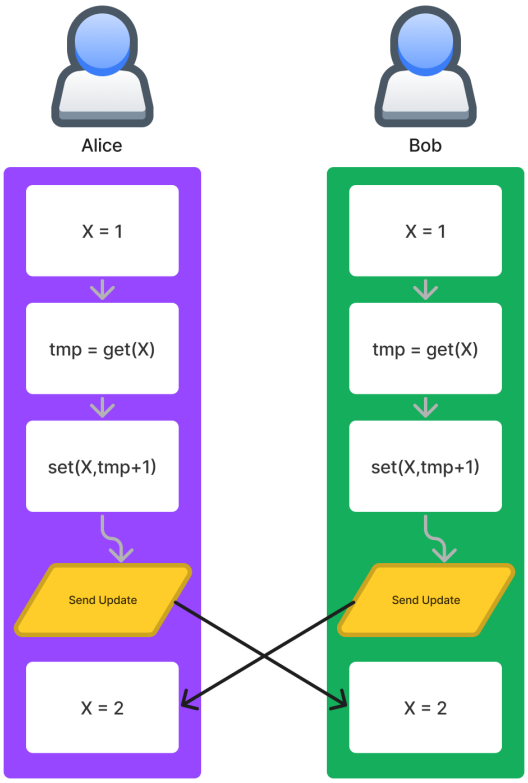


Figure 3.5: Update Limitation Example
Concurrent increments should lead to the result $X{=}3$, but this is not communicated effectively with *get* and *set* operations alone

Finally, ShelfAware Shelves do not have a local *update* operation to change a value in place atomically. For example, operation-based CRDTs can effectively handle concurrent updates, preserving concurrent counter increments across multiple users. We can approximate these updates with subsequent *get* and *set* operations. However, this operation done on other clients will not result in the correct merging of concurrent increments (Figure 3.5). Support of custom nested CRDTs would address this issue, which we will cover in the Future Work section.

Chapter 4

IMPLEMENTATION

In this chapter, we will apply our methods to a concrete implementation of the ShelfAware library. First, we will address the decision process behind our host language and describe the structure of our core Shelf and Awareness data types. Then we will examine our WebAssembly (WASM) language bindings and the specialized data structures that emerge as an extension of our core library.

4.1 Programming Language

We built ShelfAware with the Rust programming language to prioritize correctness and performance. Rust supports exhaustive compile-time verification, an expressive trait system, impressive execution times, and compatibility with other programming languages, which make it ideal for developing CRDTs. Rust's pattern-matching syntax enforces complete pattern verification at compile time. For complicated operations like merges, this feature ensures that all merge possibilities are covered as intended. Rust's trait system allows us to loosely constrain the components of our CRDT by their properties instead of requiring adherence to a specific inheritance hierarchy. The native trait definitions for Partial Ordering (*PartialEq* and *PartialOrd*), often implemented for comparison purposes on user types, allow many data types to work with ShelfAware by default. Alongside these helpful development features, we chose Rust due to its impressive performance. Rust can produce faster and more memory-efficient programs for particular tasks than C++, the industry-standard language for computing heavy applications like gaming and image processing [11].

Finally, Rust supports many libraries for binding Rust code to other programming languages. We used these capabilities to compile ShelfAware to WebAssembly and run it in the browser alongside web applications [28]. Other CRDT libraries have used these language bindings to port their data structures from Rust to Python [76] and Ruby [52]. In summary, Rust provides powerful advantages in software architecture, flexibility, and performance, making it an ideal host language for ShelfAware.

4.2 Shelf

The base data structure of ShelfAware is the Shelf. In its abstract form, the Shelf holds partially ordered data that can be sequenced by partially ordered clocks. In code, our Shelf is defined as follows:

Listing 4.1: Shelf structural definition in Rust

```
pub enum Shelf<T, MapClock, ValueClock>
where
    T: PartialEq + PartialOrd,
    MapClock: PartialEq + PartialOrd
    + PartialOrd<ValueClock> + PartialEq<ValueClock>,
    ValueClock: PartialEq + PartialOrd
    + PartialOrd<MapClock> + PartialEq<MapClock>,
{
    Value {
        value: T,
        clock: ValueClock,
    },
    Map {
```

```
        shelves: HashMap<String, Shelf<T, MapClock, ValueClock>>,
        clock: MapClock,
    },
}
```

This enumeration type defines a Shelf as a Map of Shelves or a leaf Value. The clocks associated with these options can differ, but they must be partially ordered with respect to each other and themselves. We support differing clocks for each Shelf subtype because the update process differs between the Value and Map. The merging algorithm recursively combines concurrently updated Maps, while concurrent Value updates rely on the content's type ordering to determine which update overwrites the other. During the delta update process, we only need to know the Lamport timestamp associated with Maps. However, we must have uniquely identifying clock information for the Values to determine whether the communicating clients reference identical Shelf contents, or if they have produced concurrent updates. Shared values can be pruned from the delta update, while concurrent changes must be included in the update payload. These differing requirements lead us to use simple Lamport timestamps for the Map and dot clocks or secure clocks for the Value subtype. The contents of Value must be partially ordered to resolve concurrent updates.

During instantiation, the Shelf recursively wraps the key/value mappings of the provided contents so that each Value is tracked independently. The contents of the Shelf are only mutably accessed through the *set* and *merge* methods, because the clock value is causally tied to the state of the Shelf contents. However, the contents are accessible immutably via the *get* method. Rust enforces immutability at compile time, so the Shelf contents are protected from unexpected mutations. Developers can still

clone the Shelf contents via this reference if they need to manipulate a copy of the Value.

The immutable operations and delta update process remain generic across all types of ShelfAware Shelf CRDTs. However, mutable operations require specific logic to increment each type of clock. We provide a *ClockGenerator* trait, which abstracts this logic from the Shelf implementation. *ClockGenerator*s require two methods: *new_clock()* and *next_clock(clock)*, which specify how clocks should be created and updated. We use this trait to instantiate new Shelves that contain Lamport timestamps and dot clocks.

4.3 Awareness

The Awareness structure is a specialization of the Shelf data type. It provides user-scoped read/write access to a Shelf instance and immutable access to the state of peers. In addition to the standard *get* and *set* operations, which default to the current user's scope, Awareness provides utilities to iterate over all peer Shelves or get the Shelf contents of a specific peer. This structure is ideal for sharing client-specific awareness data like cursor position, text range selection, or reactions. A typical implementation pattern for mapping awareness data to an application would be to iterate over all peers in the Awareness structure and map the internal data to UI components for presentation.

The Awareness structure can take advantage of an additional memory optimization. Since each Shelf has client-scoped write access, we can guarantee that there will never be concurrent updates to a Shelf on separate nodes. Therefore, we can use Lamport timestamps for both the Shelf Maps and Values, reducing the clocks' memory impact and comparison overhead.

Internally, Awareness stores all clients in a single Shelf, which allows it to use the delta update and merge process from the Shelf implementation.

4.4 WebAssembly Library Support

Building on the foundational library support for Rust development, we created WebAssembly bindings allowing ShelfAware to work in the browser for web applications.

We made this extension for several reasons. First, JavaScript is a common host language for CRDTs. There are more than twice as many CRDT-related repositories on GitHub written in JavaScript than any other programming language [1]. We wanted ShelfAware to be accessible to the broader developer community in the language of their choice. Furthermore, we wanted to benchmark ShelfAware against other CRDT implementations in a consistent environment. Since our competitors are browser-based CRDTs, we needed to make our system work in the browser in order for it to function as a suitable companion or substitute library. Lastly, we wanted to verify that ShelfAware could run effectively on different compile targets.

Our WebAssembly bindings open ShelfAware to a variety of uses. It can be the accompanying awareness CRDT alongside popular text-editing CRDTs like Automerge and Yjs. It can run both in the browser to track client-side interactions or as a minimalistic information storage system in Node.js server applications.

The following subsections detail the data structures provided by the WebAssembly library. Since our library is compiled before interacting with the JavaScript runtime, we collapsed our generic Shelf implementation into several defined classes. These specialized Shelf structures are the DotShelf, Awareness, and SecureShelf.

[1] Based on the combined repository results of search queries "CRDT" and "Conflict-free replicated data type" on github.com on Feb 18, 2023

4.4.1 Dot Shelf

The DotShelf is a general-purpose Shelf CRDT implementation for collaborative mutable access to the entire state of the Shelf. It stores JSON data types, using dot clocks for its leaf values and Lamport timestamps to track the state of Shelf Maps. This data structure aims to provide a thin wrapper around a Shelf implementation that can be leveraged for various applications.

4.4.2 Awareness

The Awareness class wraps the native ShelfAware Awareness structure, exposing its interface to the JavaScript library. This class would be a drop-in solution for awareness use cases. Its API structure was inspired by the Yjs Awareness API, allowing it to be used as a tool in similar environments.

4.4.3 Secure Shelf

The SecureShelf is a specialization of the Shelf structure that uses secure clocks to track leaf values and Lamport timestamps for Shelf Maps. Secure Shelves are used to safeguard the operational capacity of the Shelf CRDT in byzantine environments. Internally it uses hashes to ensure that clock and value pairs remain valid during the delta update process.

4.5 Testing

We developed a test suite to ensure the ShelfAware library operates as expected. Verifying the correctness of a CRDT can be challenging due to the number of interacting

types and nested structures. For foundational compliance, we wrote unit tests to verify the update, merge, and type ordering mechanisms. However, we suspect that our fundamental test cases did not cover edge cases that may occur in deeply nested data structures.

To cover these cases, we designed a JSON fuzzer, which could construct random JSON trees holding a vast range of values confined by parameterized uniform distributions. Fuzzers are often used in the design of programming languages to ensure that compilers can accept all types of inputs within a language specification [19]. We adopted this idea to verify that ShelfAware could accept any valid JSON input.

With this utility, we generated content for thousands of Shelf CRDT instances and performed a complete update cycle between permutations of these CRDTs. We then apply a previously integrated cached delta to the merged results. By testing the equality of these merged Shelf CRDTs, we ensure that our implementation upholds the core requirements of a CRDT: that it exchanges commutative, associative, and idempotent updates, converging to a consistent representation. These tests are publicly available in the project repository [2].

[2] `https://github.com/Waidhoferj/thesis/blob/main/shelf-crdt/src/wrap_crdt.rs`

Chapter 5

EXPERIMENTS

In this chapter, we will describe the benchmarks we developed to determine the relative efficacy of the ShelfAware framework compared to other industry-grade CRDTs. Our system is optimized for high-frequency read/write environments that require low-memory overhead for messages and local CRDT storage. Therefore, we created the memory and performance benchmark suite with these design principles in mind. Ideally, our system should be able to operate faster with less memory overhead than the competing CRDTs.

We ran all benchmarks on a 2021 MacBook Pro with an M1 Max chip.

5.1 Hypotheses

We designed these experiments to test the following hypotheses, which are based on our performance and memory impact design goals:

- The update size of ShelfAware will be smaller than other CRDTs.
- ShelfAware will have a smaller memory footprint than other CRDTs.
- ShelfAware will be able to perform local mutations faster than other CRDTs.

5.2 Competing Frameworks

We chose to benchmark ShelfAware against Automerge and the Yjs Awareness CRDT.

Automerge is a popular operation-based CRDT for text editing [29]. We wanted to verify that our implementation performed better for awareness use cases since Automerge offers a more general-purpose, feature-rich CRDT interface. If our system outperforms Automerge, we can justify the design trade-offs we made for our specialized use case.

The Yjs Awareness CRDT is a simple and effective awareness CRDT designed to accompany the featureful Yjs library [25]. It is a state-based CRDT that operates like a Last-Write Wins Register CRDT [61] , scoped to each user. This design keeps the CRDT overhead low since only a single clock is required per user to track all awareness data. However, this approach requires users to send their entire awareness state to their peers for every update. Our delta-state approach is designed to improve upon this practice, sending only the differences between client states with each update. If our system outperforms the Yjs Awareness CRDT, we demonstrate that the additional complexity of the delta state update process is a worthwhile optimization.

5.3 Performance Benchmarks

The performance benchmarks measure how quickly each CRDT can perform essential operations. The performance of these operations is crucial in our awareness use case, where the application may update and sample awareness data at millisecond frequency to update the user interface. Slow operations may delay the UI rendering process, leading to frustrating periods of inactivity for the user.

We measured performance in operations per second. More efficient programs will be able to perform more operations per second. We developed the following tests to benchmark each CRDT:

- Insertion Performance: The speed at which the CRDT can perform nine thousand top-level key-value insertions.

- Merge Performance: The speed at which the CRDT can encode, decode, and merge a batch of updates. To promote a large delta, we merge values from a larger randomly generated JSON tree into a smaller tree. We initialize a *small-Fuzzer* with a depth range of 2-4 and a branch range of 0-3. The *largeFuzzer* is initialized with a depth range of 3-5 and a branch range of 1-5.

To keep the benchmarking environment consistent, we compared our WebAssembly DotShelf implementation to the competing CRDTs in a Node.js runtime using JavaScript's standard performance module. We cycled each benchmark three times as a warm-up round to stabilize cache coherence before recording results. The final test results are an average of seven simulations to measure expected performance. Each simulation allows for an initialization phase before measuring the critical code juncture for performance. Initialization involves setting each CRDT with random JSON values so that the benchmarks emulate a system in active use. We hold these initial values constant across each simulation and between CRDTs in the same benchmark. The warm-up rounds and simulations occur sequentially within each CRDT to prevent any influence from uncollected heap allocations marked by other contenders.

5.4 Memory Benchmarks

The memory benchmarks measure the size of the CRDT and its updates during different use case scenarios. Measuring memory usage is essential because developers integrate CRDTs into local-first software, which means the system needs to work on many different hardware devices. Mobile devices, for example, are memory constrained. These benchmarks provide insight into how efficiently each CRDT uses

the host's memory resources. Each benchmark compares the size of the target data structure in bytes. To determine the size of CRDT update packages, we encode the update into a byte-array and measure the array size.

To compare the full size of the CRDT, we use the *object-sizeof* JavaScript utility library. This library estimates the size of JavaScript data structures by serializing them into JSON and estimating the size of each data type. Unfortunately, some structures are untraceable by the library, so we performed the following adaptations to improve the accuracy of the memory estimation:

- We call the children of JavaScript Maps explicitly with *object-sizeof*, since the library ignores them by default. Without this approach, CRDTs appear to be smaller than the values they contain, which is impossible.
- The size of values stored in WASM are calculated with Rust's standard data structure size calculator [64]. Without this approach, WASM-based classes appear minuscule since JavaScript cannot directly traverse all properties on WASM data structures.

Table 5.1: JSON Fuzzer Parameters

Parameter	Range
Depth	3..5
Branches	1..4
Leaves	300..500

JSON fuzzer parameters used to generate random JSON trees for performance and memory benchmarks.

We tested the memory usage of each CRDT under the following scenarios:

- Size After Deletion: The size of the CRDT (including metadata and data) and delta update size after all data has been deleted. Size after deletion shows the

prunability of the CRDT. Ideally, deleted values should not contribute to the CRDT's metadata overhead.

- Random Update Size: The size of the delta update between two randomly generated CRDTs as an encoded byte array. This test emulates a scenario where a client requests an update after being disconnected from the network for an extended period.

- Single Update Size: The size of the delta update between two CRDTs that differ by only a single value. This benchmark exemplifies the memory impact of update passing in a low-latency network. Awareness updates are small, rapidly changing data packages, similar to the updates in this test.

We performed each benchmark for one thousand iterations. At every iteration, the CRDTs were prepopulated with a randomly generated JSON tree from our JSON fuzzer. We seeded the fuzzer so that each CRDT was tested with the same randomly-generated sequence of trees. The fuzzer generates trees by sampling from a provided range of values to determine the depth, level-wise branching factor and level-wise leaf count. See Table 5.1 for details about the fuzzer parameterization.

Chapter 6

RESULTS

In this chapter, we discuss the results of the experiments outlined in the previous chapter. We will begin by comparing the performance and memory impact of our DotShelf implementation to Automerge and Yjs Awareness. Then we will run the same set of benchmarks on each specialized Shelf type in our WebAssembly library (DotShelf, Awareness and SecureShelf) to determine the trade-offs of using each implementation.

6.1 Memory Benchmark Analysis

This section discusses the results of the memory-related benchmarks. These tests focused on the size of the CRDT in memory, and the updates passed between the CRDTs.

Table 6.1: Memory Benchmark Results

	CRDT		
Benchmark	Automerge	Yjs Awareness	DotShelf (Ours)
Update Size - Single Change (avg B)	122	127491	69
Update Size - Complete Deletion (avg B)	121	6	56
Update Size - Random Merge (avg KB)	74.3	127.4	254.9
CRDT Size - Random JSON (KB)	18873.8	537.3	495.0
CRDT Size - Complete Deletion (KB)	4376.9	48.7	0.034

6.1.1 Update Size

Our Update Size benchmarks tested the byte length of encoded updates from each CRDT during a complete content deletion, a single change, and a merging to two

heterogeneous JSON trees. The results of these tests can be found in Table 6.1. We summarize our findings below:

- ShelfAware had half the delta size as Automerge for the complete deletion and single change test variants. Due to its feature-rich interface, Automerge updates require more information for minor updates than our ShelfAware CRDT. For awareness use cases with rapid, incremental updates, application programmers can leverage ShelfAware to reduce the total network communication load.

- Yjs Awareness has a lower memory overhead during complete deletion than ShelfAware. This advantage is likely due to the efficiency of the Yjs update representation. ShelfAware deltas include structural information about the CRDT that is extraneous for complete deletion operations.

- ShelfAware has a fractional single change cost compared to Yjs Awareness. Since Yjs Awareness uses a classic state CRDT, it must include all stored values in every update. Therefore, large, frequently updated state trees can drastically increase the number of network packets that clients need to share. In contrast, ShelfAware communicates updates as small deltas, which consume far fewer network resources.

- ShelfAware produces larger updates when merging two mostly divergent CRDTs. This result stems from several key architectural differences. ShelfAware relies on deltas to keep the memory impact of its updates small. However, two divergent Shelves must share most of their state, negating this advantage. Furthermore, ShelfAware keeps track of each element with a clock, which introduces additional overhead compared to the single clock system of Yjs awareness. Additionally, Yjs and Automerge use specialized run-length encoding schemes to reduce the size of their delta updates. ShelfAware uses a standard binary encoding system

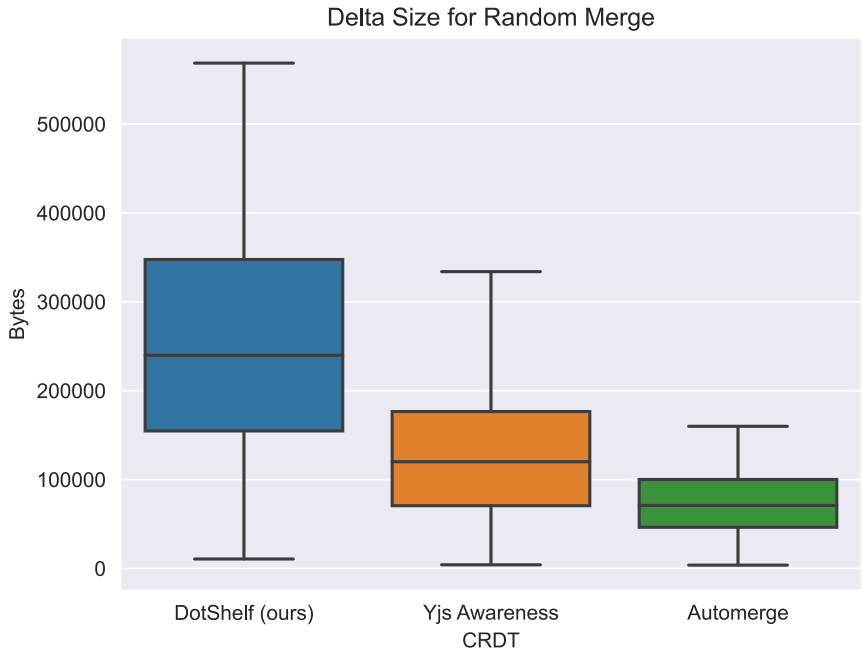


Figure 6.1: Random Merge
Update size of a merge between two randomly generated CRDTs

without any domain-specific optimizations. A specialized encoding scheme for ShelfAware could reduce the size of large delta updates.

Reflecting on Hypothesis 1, we can confirm that ShelfAware produces the smallest single change delta updates. Therefore, in this area, we can accept the hypothesis. However, Yjs produces smaller complete deletion updates than ShelfAware. Additionally, ShelfAware produces the largest update size while merging heterogeneous states. Therefore, we must reject the hypothesis in these scenarios. Heterogeneous merging and complete deletion occur when clients come online and offline. These are one-time operations that begin and end the user lifecycle, while single change updates frequently occur throughout the user experience. This frequency difference means that ShelfAware will be the optimal choice for reducing the size of most updates passed through the network.

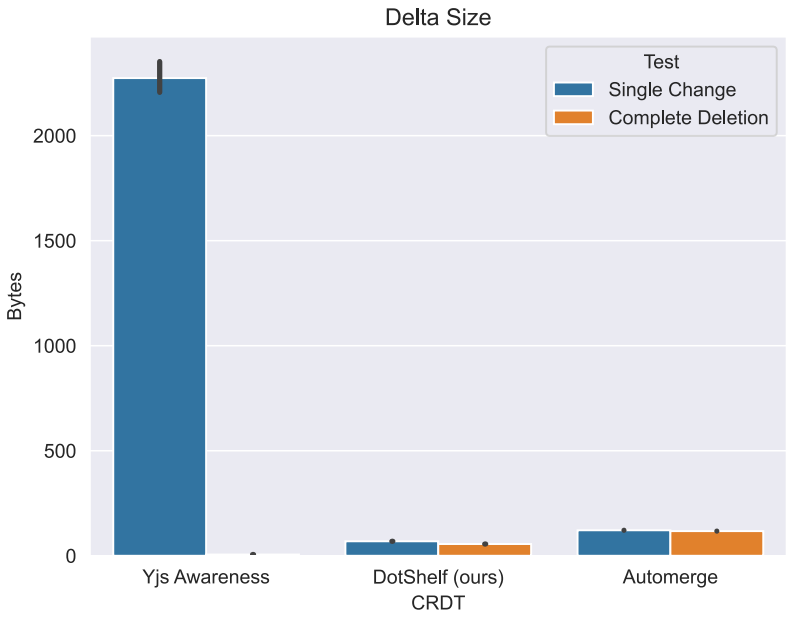


Figure 6.2: Single Update
Size of message between CRDTs after a single change and after a complete content deletion. Smaller initialization data was used to keep the Yjs update size from overshadowing the scale difference between Automerge and DotShelf.

6.1.2 CRDT Size

Our CRDT Size benchmarks track the memory footprint of each contender's data structures. We examined the size of each contender after initialization with a randomly generated JSON tree and the size after complete content deletion. Table 6.1 details our findings. We provide an analysis below:

- ShelfAware has less metadata overhead than the other CRDTs. Yjs Awareness works in tandem with a Yjs Document, which adds to the size requirement of the CRDT. Automerge keeps a record of past updates in a dependency graph, which makes its metadata overhead grow over time. The primary metadata cost of a ShelfAware CRDT is the clock associated with each value, which is approximately the size of two integers.

- ShelfAware is far smaller after a complete deletion than other CRDTs. Unlike Automerge, ShelfAware is completely pruneable so the CRDT can erase any metadata associated with the removed data. As a result, ShelfAware can adjust its memory usage proportionally to the data it represents instead of the number of updates it receives.

Based on our findings, we can accept Hypothesis 2. ShelfAware proved to be the most space-efficient candidate when representing a random JSON tree and the smallest upon complete deletion of its contents. The sizable difference in memory overhead between a feature-rich CRDT like Automerge allows ShelfAware to be a low-impact companion to Automerge for communicating awareness information.

We advise the reader to approach our measurements of the Automerge CRDT size with skepticism. The memory size we report reflects the analysis from *object-sizeof*, a widely used JavaScript memory measurement package. However, we suspect that cyclic references within the cache structure of Automerge are causing objects to be counted multiple times during the memory analysis. With this in mind, the actual CRDT size for Automerge is likely much smaller. Even so, the lineage graph of the Automerge system necessitates more data storage than our CRDT, which stores only the latest values for each Shelf.

6.2 Performance Benchmark Analysis

Table 6.2: Performance Benchmark Results

	CRDT		
Benchmark	Automerge	Yjs Awareness	DotShelf (Ours)
Inserts (ops/sec)	30	74	67132
Merges (ops/sec)	2903	47080	134366

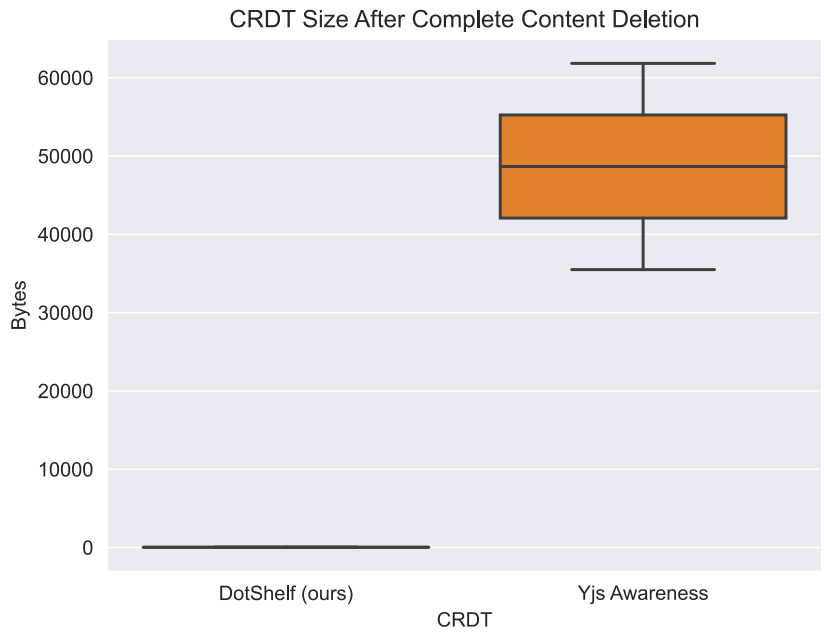


Figure 6.3: CRDT Size After Deletion
Size difference between Yjs Awareness and our DotShelf after complete deletion of a large JSON tree of contents. Automerge is excluded due to its large scale.

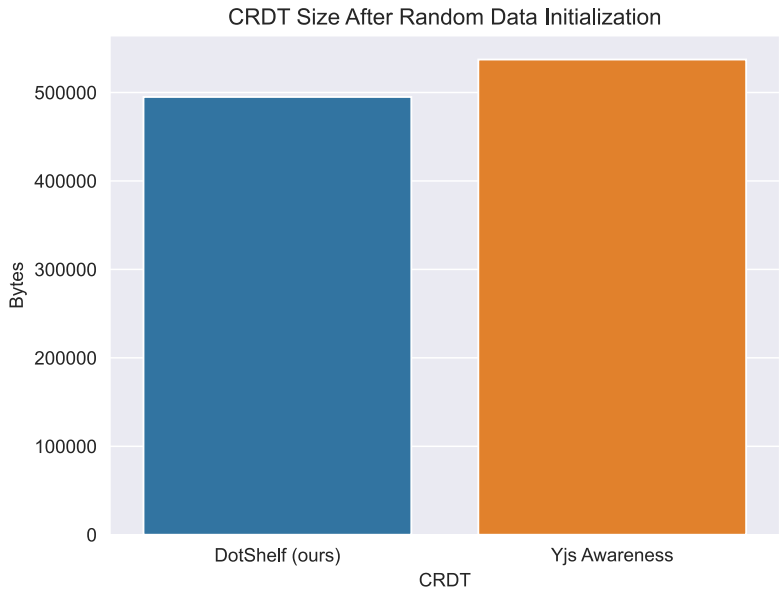


Figure 6.4: CRDT Size
Size difference between Yjs Awareness and our DotShelf after being initialized with the same randomly generated JSON tree. Automerge is excluded due to its large scale.

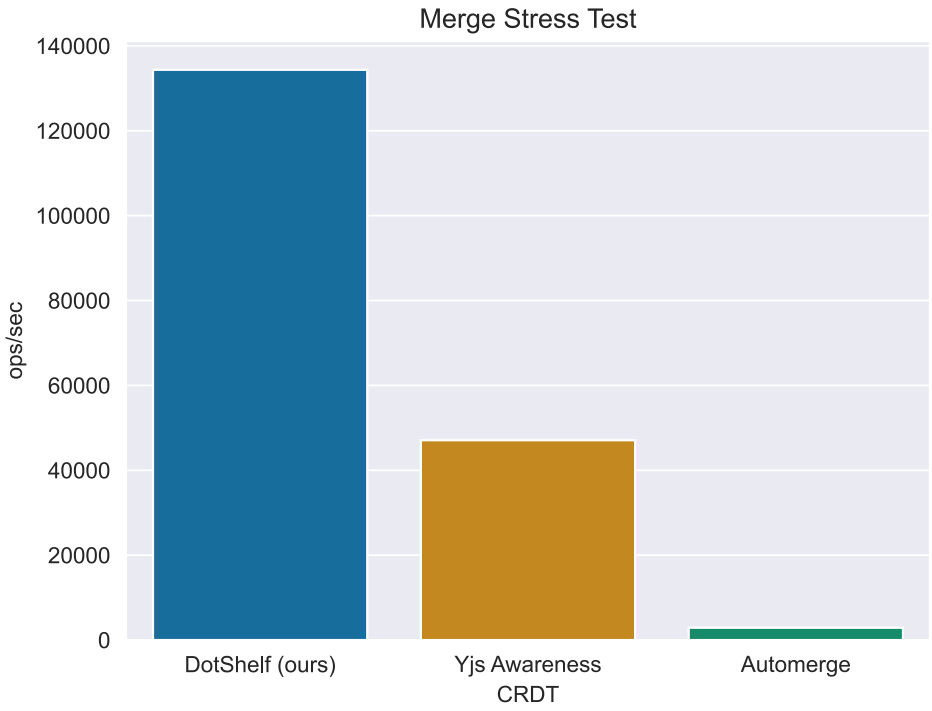


Figure 6.5: Merge Benchmark
Performance of merging updates locally between two nodes

In this section, we review the mutation performance of each CRDT, specifically the *merge* and *set* operations. We summarize our findings below:

- ShelfAware was four times faster during *merge* operations than the other CRDTs.
- ShelfAware was orders of magnitude faster during *set* operations than the other CRDTs.

Based on these findings, we can confirm Hypothesis 3. ShelfAware provides a faster interface to update data than the other CRDTs. Several possible causes exist for the significant performance difference between ShelfAware and the other CRDTs. First, ShelfAware executes most of the business logic inside the WebAssembly runtime. As a result, ShelfAware Shelves take advantage of the performance of the Rust programming language, which can be nine to thirteen times faster than pure JavaScript while

completing computationally expensive operations [40]. Second, ShelfAware operates in a stricter environment than both other CRDTs. WebAssembly bindings ensure that ShelfAware takes complete ownership of an object when passed to the CRDT. After the data has been copied to the WASM runtime, we can pass ownership of that data between Shelves without a reference counter while ensuring exclusive access. In contrast, Yjs Awareness holds non-exclusive references to internal data. When it performs a *set* operation, Yjs Awareness initializes a new copy of the parent object, which requires an $O(N)$ copy of the child references. Adding a new key under this paradigm can be computationally expensive for large flat objects. Third, the state delta process necessitates fewer operations during the merge process since only a subset of the data is sent, deserialized, and compared to the local Shelf.

6.3 ShelfAware Internal Benchmarks

In this section, we will test the three ShelfAware data structures from the WebAssembly library using the performance and memory benchmarks from the previous section. We will discuss how the optimizations made for each data structure are reflected in the results.

Table 6.3: Internal ShelfAware Benchmark Results

	CRDT		
Benchmark	Awareness	Dot Shelf	Secure Shelf
Inserts (ops/sec)	58216	67132	61922
Merges (ops/sec)	91221	134366	67820
CRDT Size - Random JSON (KB)	440	494	604
CRDT Size - Complete Deletion (B)	54	34	34
Update Size - Complete Deletion (avg B)	85	56	56
Update Size - Random Merge (avg KB)	220	254	254
Update Size - Single Change (avg B)	90	69	69

6.3.1 Performance Benchmarks

The Dot Shelf was the most performant for both merges and insertions. This result is to be expected because the other data structures are based on the DotShelf with additional complexities. For example, the Awareness structure tracks client information, adding an additional layer of indirection. Even so, the Awareness structure has a partial advantage in clock comparison since all of its clocks are simple Lamport timestamps. The Secure Shelf must hash its contents before merging, adding additional operations to each integration.

While Awareness outperformed the Secure Shelf in merges, the Secure Shelf had a faster element insertion speed. This result shows that the additional layer of client indirection can produce a noticeable difference during rapid insertions on a shallow Shelf data structure.

Merge performance was the most variable across the Shelves, while the spread of insertion performance was minimal. A key takeaway is that adding clock hashing to support the Secure Shelf can cut the merge performance in half. If the deployment environment is secure, there is a significant performance advantage in choosing the Dot Shelf over the Secure Shelf.

6.3.2 Memory Benchmarks

The CRDT size benchmarks illustrate the trade-offs of the Awareness structure. For the Random JSON initialization, Awareness had the smallest memory footprint. This finding is partly due to the structure's clock size being half its competitors' size. Therefore, Awareness's overall memory footprint will grow slower than the size of Shelf data structures with more complex clocks. However, during the CRDT Size

Complete Deletion test, the additional client overhead of Awareness made the default size larger than its competitors. Fortunately, this is a fixed memory cost attributed to the client identifying information stored on each instance. The Secure Shelf performed identically to the Dot Shelf after content deletion, but the larger hash clocks added apparent bulk during the Random JSON test. Therefore, a space cost is associated with using Secure Clocks compared to dot clocks.

The update size benchmarks showed a similar trend. The Awareness structure produced a smaller update for the larger Random Merge delta, partially due to its smaller clock size. However, it produced larger delta state updates for Complete Deletion and Single Merge due to the additional client-related information attached to these updates. Surprisingly, the Dot Shelf and the Secure Shelf had identical update sizes for all three delta tests. This result is strange, given the apparent difference in clock sizes in the CRDT Size tests.

In summary, the Awareness structure is the most scalable ShelfAware implementation, with a small fixed overhead. However, the Secure Shelf has an additional memory cost, one which does not transfer to its delta updates. The Dot Shelf performs similarly to the Secure Shelf, with a lower memory impact when it contains smaller values.

6.4 Conclusion

We present ShelfAware as an efficient solution for communicating awareness in collaborative applications. Our system prioritizes performance and memory by applying critical optimizations, including pruning and delta-state updates. In addition, we offer a standalone open-source library written in Rust for developers to apply these concepts in their applications. Atop the library, we provide WebAssembly bindings that enable developers to leverage ShelfAware data types in the browser. The ShelfAware

ecosystem provides a Shelf CRDT implementation with modular value and clock components. We demonstrate extensions of this data structure with Awareness and Secure Shelf implementations for specialized use cases. Finally, we present a benchmark analysis of these data structures compared to production-grade CRDTs and demonstrate the advantages of our ShelfAware system.

6.5 Future Work

There are several improvements that we can make to our Shelf CRDT. First, we want to extend the Shelf implementation to support nested CRDTs through an update and merge interface. We are interested in how this feature would enable developers to use the Shelf as a building block for more complex CRDT implementations. We want to explore a specialized encoding schema using run-length encoding to reduce the size of our delta updates. Additionally, further testing in a distributed cluster environment would provide more insight into how this system would behave in a production setting. The Shelf interface could benefit from different update strategies, including an eager delta system that records updates as they occur in order to compile delta Shelves without performing a state vector comparison. Finally, we are interested in exploring parallelization techniques to improve the performance of the update process.

BIBLIOGRAPHY

[1] A. Acar, H. Aksu, S. Uluagac, and M. Conti. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. 51.

[2] P. S. Almeida, A. Shoker, and C. Baquero. Delta state replicated data types. 111:162–173.

[3] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. 36(4):335–371.

[4] A. Auvolat and F. Taïani. Merkle Search Trees: Efficient State-Based CRDTs in Open Networks. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 221–22109.

[5] M. Barbosa, B. Ferreira, J. Marques, B. Portela, and N. Preguiça. Secure Conflict-free Replicated Data Types. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking*, ICDCN '21, pages 6–15. Association for Computing Machinery.

[6] Basho Technologies. Why Riak KV?

[7] A. Brocco. Delta-State JSON CRDT: Putting Collaboration on Solid Ground. In C. Johnen, E. M. Schiller, and S. Schmid, editors, *Stabilization, Safety, and Security of Distributed Systems*, Lecture Notes in Computer Science, pages 474–478. Springer International Publishing.

[8] A. Brocco. The Document Chain: A Delta CRDT framework for arbitrary JSON data. page 12.

[9] A. Brocco. Melda: A general purpose delta state JSON CRDT. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '22, pages 1–7. Association for Computing Machinery.

[10] R. Brown. Vector Clocks Revisited Part 2: Dotted Version Vectors.

[11] W. Bugden and A. Alahmar. Rust: The Programming Language for Safety and Performance.

[12] M. Caderek. Benny - a dead simple benchmarking framework.

[13] W. Cai, F. He, and X. Lv. Multi-core accelerated CRDT for large-scale and dynamic collaboration.

[14] J. Day-Richter. What's different about the new Google Docs: Making collaboration fast.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. 41(6):205–220.

[16] Ditto. Ditto - Sync without Internet.

[17] S. Dunham. Apple Notes Export Tools.

[18] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 399–407. Association for Computing Machinery.

[19] J. Fell. A Review of Fuzzing Tools and Methods. page 21.

[20] P. Frazee. Pfrazee/crdt_notes at 68c5fe81ade109446a9f4c24e03290ec5493031f.

[21] V. K. Garg. *Introduction to Lattice Theory with Computer Science Applications*. John Wiley & Sons.

[22] S. Gentle. 5000x faster CRDTs: An Adventure in Optimization.

[23] L. Hupel. An introduction to Conflict-Free Replicated Data Types.

[24] K. Jahns. Are CRDTs suitable for shared editing?

[25] K. Jahns. Awareness.

[26] K. Jahns. CRDT benchmarks.

[27] K. Jahns and G. Little. Braid meeting-8.

[28] S. M. Jain. WebAssembly with Rust and JavaScript: An Introduction to wasm-bindgen. In S. M. Jain, editor, *WebAssembly for Cloud: A Basic Guide for Wasm-Based Cloud Apps*, pages 57–85. Apress.

[29] M. Kleppmann. Automerge: A new foundation for collaboration software.

[30] M. Kleppmann. CRDTs: The Hard Parts.

[31] M. Kleppmann. A Critique of the CAP Theorem.

[32] M. Kleppmann. Making CRDTs Byzantine Fault Tolerant. page 8.

[33] M. Kleppmann and P. Alvaro. Research for practice: Convergence. 65(11):104–106.

[34] M. Kleppmann and A. R. Beresford. A Conflict-Free Replicated JSON Datatype. 28(10):2733–2746.

[35] M. Kleppmann and H. Howard. Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases.

[36] M. Kleppmann and rae. How Automerge works — Automerge CRDT.

[37] S. A. Kollmann, M. Kleppmann, and A. R. Beresford. Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing. 2019(3):210–232.

[38] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical Physical Clocks. In M. K. Aguilera, L. Querzoni, and M. Shapiro, editors, *Principles of Distributed Systems*, Lecture Notes in Computer Science, pages 17–32. Springer International Publishing.

[39] H. Kwong. 7.4: Partial and Total Ordering.

[40] K.-I. D. Kyriakou and N. D. Tselikas. Complementing JavaScript in High-Performance Node.js and Web Applications with Rust and WebAssembly. 11(19):3217.

[41] lakshita. Introduction to Merkle Tree.

[42] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. 21(7):8.

[43] G. Litt, S. Lim, M. Kleppmann, and P. van Hardenberg. Peritext: A CRDT for Rich-Text Collaboration.

[44] G. Little. Shelf.

[45] G. Little. Shelf: A remarkably small, remarkably useful CRDT.

[46] J. Long. CRDTs for Mortals - James Long at dotJS 2019.

[47] G. Lopez and L. A. Guerrero. Awareness Supporting Technologies used in Collaborative Systems: A Systematic Literature Review. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and*

Social Computing, CSCW '17, pages 808–820. Association for Computing Machinery.

[48] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo. Optimizing Bloom Filter: Challenges, Solutions, and Comparisons. 21(2):1912–1949.

[49] X. Lv, F. He, W. Cai, and Y. Cheng. A string-wise CRDT algorithm for smart and large-scale collaborative editing systems. 33:397–409.

[50] N. Masinde and K. Graffi. Peer-to-Peer-Based Social Networks: A Comprehensive Survey. 1(5):299.

[51] S. E. McDaniel and T. Brinck. Awareness in collaborative systems. In *CHI '97 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '97, page 237. Association for Computing Machinery.

[52] H. Moser. Y-crdt/yrb: Ruby bindings for yrs.

[53] Neso Academy. Lattice.

[54] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. pages 39–49.

[55] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma. Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types. In P. Cimiano, F. Frasincar, G.-J. Houben, and D. Schwabe, editors, *Engineering the Web in the Big Data Era*, Lecture Notes in Computer Science, pages 675–678. Springer International Publishing.

[56] K. Niemantsverdriet, H. V. Essen, M. Pakanen, and B. Eggen. Designing for Awareness in Interactions with Shared Systems: The DASS Framework. 26(6):1–41.

[57] G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for P2P collaborative editing. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, CSCW '06, pages 259–268. Association for Computing Machinery.

[58] A. Pandey, A. Bieniusa, and M. Shapiro. Persisting the AntidoteDB Cache: Design and Implementation of a Cache for a CRDT Datastore.

[59] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. A Commutative Replicated Data Type for Cooperative Editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403.

[60] N. Preguiça. Conflict-free Replicated Data Types: An Overview.

[61] N. Preguiça, C. Baquero, and M. Shapiro. Conflict-free Replicated Data Types (CRDTs).

[62] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, CSCW '96, pages 288–297. Association for Computing Machinery.

[63] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. 71(3):354–368.

[64] RustLang. Size_of in std::mem - Rust.

[65] M. Régner. Group mode.

[66] A. Scandurra, N. Sobo, J. Rudolph, and A. Bayer. Teletype-crdt.

[67] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types.

[68] B. Sypytkowski. CRDT optimizations.

[69] B. Sypytkowski. An introduction to state-based CRDTs.

[70] B. Sypytkowski. Optimizing state-based CRDTs (part 2).

[71] B. Sypytkowski. Shelf: Easy way for recursive CRDT documents.

[72] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. `www.openssl.org`, April 2003.

[73] TipTap. Collaborative editing – Tiptap Editor.

[74] M. Toomim. What makes a Braid? Merges of Patches on Ranges. page 21.

[75] L. Wagerfield. Conflict-free Replicated Data Types.

[76] J. Waidhofer, K. Jahns, M. Kafonek, and D. Brochart. Y-crdt/ypy.

[77] E. Wallace. How Figma's multiplayer technology works.

[78] G. Weldesselasie and K. Jahns. Braid meeting-13.

[79] Wikipedia. Lamport timestamp.

[80] D. Yang, D. Tian, J. Gong, S. Gao, T. Yang, and X. Li. Difference Bloom Filter: A probabilistic structure for multi-set membership query. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6.

[81] Y. Zhang, H. Wei, and Y. Huang. Remove-Win: A Design Framework for Conflict-free Replicated Data Types. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 607–614.