UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Supporting Undo and Redo for Local-First Software

An Implementation in Automerge

Eric Brattli

INF-3981 Masters Thesis in Computer Science June 2021

Abstract

With the advance of cloud computing and centralization of data, a new effort goes entirely in the other direction and aims for decentralization of data through local-first software. Collaborative applications created this way need strong undo and redo support to handle the inevitable mistakes that take place in a collaborative setting.

Local-first software can be effectively built using Conflict-Free Replicated Data Types (CRDTs), where all the application data is stored locally at the user.

This paper presents the design and implementation of a new approach for undoing and redoing operations in a modern open-source operation-based CRDT. The current approach is severely limited and can only undo local operations. Our approach allows for generic and selective undo and redo for consistently replicated registers.

Contents

List of Figures

List of Tables

1 Introduction

While cloud computing continues to centralize our data in the cloud, where our data will be wholly-owned and controlled by other companies, local-first software takes a different path and stores all user data locally at the user's computer. Updates to the user's data modify the local data first before sharing its data with the application or peers in a network. Storing the data locally at the user gives the user complete control and ownership of his data.

When operating with local-first software and there is a need for shared data between users, there is also a need for consistency. There are several ways of achieving this, with the most prominent being Operational Transform (OT) and Conflict-Free Replicated Datatypes (CRDT). OP has long been commonly used for solving consistency; however, as OP can be very complicated to implement correctly, there has been an increased usage of CRDTs.

CRDTs are often used for collaborative applications, for example, for creating collaborative editing programs. With many users editing the same data structure, there will often be conflicts and mistakes where users will want to undo and redo their edits.

Automerge is a JSON-CRDT which means that it has a similar structure as that of a JSON document, while it also possesses CRDT capabilities which allow it to be shared among users. As we will see in chapter 3, Automerge has limited undo and redo capabilities. In this paper, we design and implement generic undo and redo for the register implementation of Automerge.

2 Technical Background

This section explains the background information on CRDTs, the different types of CRDTs, and current work on generic undo on CRDTs. First, we will explain what a join semilattice is, which is the background for state-based CRDTs.

2.1 Join Semilattice

We first provide the definitions for what a poset is and what a least upper bound is, and then we can define a join semilattice [1] using those definitions.
Definition 2.1.1. *A partially ordered set is a set X with a binary relation $\leq$ where the following conditions hold for all a, b and c in X*
1. $a \leq a$ X is reflexive.
2. if $a \leq b$ and $b \leq a$, then $a = b$ X is antisymmetric.
3. if $a \leq b$ and $b \leq c$, then $a \leq c$ X is transitive.
Definition 2.1.2. *The least upper bound of b and c is an element where $a \geq b$ and $a \geq c$, and there is no other element a' where $a' \leq a$ and $a' \geq b$ and $a' \geq c$.*
Definition 2.1.3. *A join of two elements in a partially ordered set is their least upper bound.*
Definition 2.1.4. *Join semilattice is a partially ordered set where all pairs of elements have a least upper bound.*

Adding to this definition, we define a monotonic join semilattice.

Definition 2.1.5. *A monotonic join semilattice is a join semilattice where merging two states in the semilattice computes the least upper bound of the two states. Also, all updates applied to a state are inflationary, meaning that a state always monotonically increases when it applies new updates.*

2.2 Strong Eventual Consistency

Strong Eventual Consistency (SEC) [2] ensures that two nodes that have received the same updates also have the same state. As its name suggests, SEC is a more substantial property than eventual consistency, and it is a crucial property of CRDTs. We start by defining eventual consistency.
Definition 2.2.1. *A distributed system of replicated data that satisfies the following three properties is eventually consistent.*
Eventual delivery: *If an update is delivered to one replica, it is eventually delivered to all replicas.*
Convergence: *Replicas that have received the same updates will eventually converge to the same state.*
Termination *All method executions terminate*

Strong Eventual Consistency adds a stronger convergence constraint to eventual consistency.
Definition 2.2.2. *A system has strong eventual consistency if it is eventually consistent, and replicas that have received the same updates have an equivalent state. This property is called strong convergence.*

From these definitions, we can see that SEC ensures that all replicas with the same updates also have the same state. EC does not guarantee this since two replicas with the same updates might need to communicate between themselves to arrive at the same state after some time. That is, EC may need coordination to resolve conflicts.

2.3 CRDTs

CRDTs are data structures that are replicated across multiple nodes. Each replica is updated locally, and at specific points, the replicas communicate their changes to other replicas. When a node receives updates from other nodes, it merges the new changes into its local data structure. No consensus algorithm is needed when communicating new changes. All CRDTs have strong eventual consistency as defined above, ensuring that replicas have an equivalent state when they have received the same updates.

2.3.1 State-based CRDTs

A state-based CRDT is a monotonic join-semilattice, as defined earlier. This property also ensures that it is strongly eventually consistent [2]. When a replica in a state-based CRDT communicates its updates with another replica, it sends its entire state to the other replica. The other replica will then receive the other state and merge it with its own. The merge function is both commutative, associative, and idempotent. These properties ensure that updates do not need to arrive in any particular order, and they can arrive multiple times. Replicas will then have an equivalent state as long as they have received the same updates in any order.

One of the downsides of state-based CRDTs is sending the entire state to other replicas when communicating their updates. In many data structures, the state can be of significant size, meaning that they will often send a large amount of data. For this reason, state-based CRDTs are more often used in file systems and databases, where updates are infrequent and where updates often change large parts of the data structure. When updates are frequent and small

2.3.2 Delta State-based CRDTs

Delta state-based CRDTs are similar to state-based CRDTs; however, they recognize the main problem with state-based CRDTs: they have to send their entire state to other replicas when they want to communicate updates. Delta state-based CRDTs improve on state-based CRDTs in that replicas only send the join-irreducible state that the other replica has not seen yet. We define the join-irreducible state and the join decomposition of a state below.
Definition 2.3.1. *Join irreducible state is a state that cannot be the join of any other states except for itself. In other words, we can regard join-irreducible states as primitive states.*
Definition 2.3.2. *The join decomposition of a state is the set of join-irreducible states that compose the state when composed together. [3]*

When two replicas in a delta state-based CRDT wish to communicate their updates, they will compute the join decomposition of their state and figure out the smallest set of join-irreducible states that they can send to each other. They will then communicate only the join-irreducible states that the other node is missing.

2.3.3 Operation-based CRDTs

Operation-based CRDTs change their state through operations that specify a change to the state. When replicas communicate their latest changes to other replicas, they only communicate the latest operations that they have received. Comparable to delta state-based CRDTs, this gives them the advantage that they do not have to send the entire state when communicating.

A critical difference between operation-based CRDTs and the two other approaches discussed above is that operation-based CRDTs require causal delivery of messages. This difference means that operation-based CRDTs need a causal broadcast channel or a deterministic way to determine the order of concurrent messages.

2.3.4 Similarities Between Different Type of CRDTs

State-based, delta state-based and operation-based CRDTs are considerably similar and can emulate each other. We can see that an operation-based CRDT can have operations that send join-irreducible states, in which case we would have an operation-based CRDT that emulates a delta state-based CRDT. We could also create an operation that contains the entire state and merges it, in which case we would have emulated a state-based CRDT. Paper [2] shows that op-based and state-based are equivalent with respect to supporting SEC.

This insight is helpful since it allows us to translate solutions for one type of CRDT to another type of CRDT.

2.4 Undo

Undo and subsequently redo are two important functionalities in collaborative systems. Users will often make mistakes that they will want to undo, and they will also want to undo the mistakes of the other users with whom they are collaborating. A common problem regarding undo in collaborative environments is whether a user should undo other users edits, called global undo, or if they should only undo their own, called local undo[4]. It is more common for systems to offer local undo as it is easier to implement and can be considered more intuitive for some systems.

Another common consideration that is important to make in a collaborative system is whether one can only undo the latest edits or if one should be able to undo any edit. The ability to undo any previous edit is called selective undo.

2.5 Undo-length

One way of implementing undo and redo on delta state-based CRDTs is by using undo-lengths as first described by Weihai [5]. The undo-length is an integer that is zero for normal changes, and we increment it each time we undo/redo that change. For example, if we add **a = 10** to a register CRDT, then that change has an undo-length of 0. When we undo that change, then the undo-length is incremented to 1, and then to 2 when we redo that change again. We show an example in figure 2.1

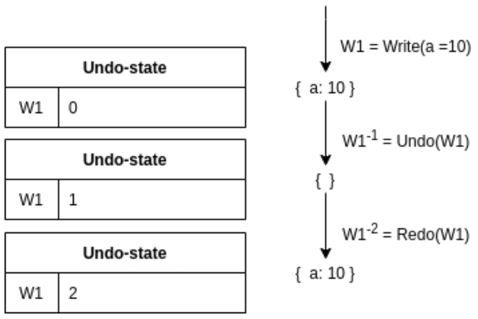

Figure 2.1: Example of undo-state CRDT

Undo-length is a part of the undo-state CRDT that can be augmented with any other delta state-based CRDT to provide generic undo support. The undo-state CRDT works by registering join-irreducible states and mapping them to their undo-length. We can merge two undo-states by merging the maps of join-irreducible states to their undo-length and then choosing the highest undo-length if the join-irreducible state exists in both maps.

From the map of join-irreducible states to undo-lengths, we can build the entire state of the CRDT by joining the join-irreducible states where the undo-length is an even number. If it has an even number, then the state has either not been undone or undone and then redone again. If the undo-length is odd, then the join-irreducible state is currently undone, and we should not use it to build the entire state.

3 Automerge

Automerge [6] is an open-source operation-based CRDT that is implemented in JavaScript. It presents a similar interface as a JSON document that can be shared and collaborated on between users. We start by explaining some common terminology used when discussing Automerge.

3.1 Terminology

Node is an independent process that runs Automerge. A distributed system can consist of many nodes on different physical computers that collaborate with Automerge. However, we can also run several nodes on a single computer that all collaborate on Automerge.

Actor Id is the unique identification of a node in Automerge.

Object Id is the unique identification of an object in Automerge. Ordinary objects in Automerge are maps, lists, tables, and text. Objects will often contain other objects.

Sequence number is a number that identifies a change performed at a specific node. The sequence number starts at 1 for the first change, 2 for the second, and increments for each change.

Vector clock is a map of actor id to its sequence number. All nodes keep track of the latest sequence number they have received from each node, and they keep track of it in a vector clock.

Dependencies is a list of changes from different nodes on which a single change depends. Nodes store these dependencies as a vector clock.

Operation An operation is a single modification on an Automerge object. For example, it can be to create an object or set a property on an object.

Change A call to `Automerge.change` can make multiple modifications to an Automerge document, where it can change several objects. These modifications are all recorded as multiple operations that are stored in a single change. A change applies multiple operations atomically.

3.2 Interface

In this section, we show an overview of the interface of Automerge in order to understand how we can use it.

Function	**Description**
`init`	Creates an initial empty document.
`from`	Initializes a document from a JavaScript object.
`change`	Makes a change to the document through a callback function.
`undo`	Undoes the last local operation on the document.
`redo`	Redoes a previously undone operation on the document.
`load`	Loads a document from JSON format.
`save`	Allows saving a document by returning it in JSON format.
`merge`	Merges changes from another node.
`getChanges`	Gets changes from an updated document compared to an older version.
`getHistory`	Gets all changes from a document since the beginning.
`applyChanges`	Applies a list of changes to the document.

Table 3.1: Automerge Interface

Nodes collaborating with Automerge will start by creating their empty local document using `Automerge.init`, and then they will start making changes to the document using `Automerge.change`. They can then get a list of the changes they have made using `Automerge.getChanges`, then they send those changes to other nodes that can merge them using `Automerge.applyChanges`. We will see more examples of this process in this chapter.

3.3 Using Automerge

In this section, we will show some examples of how to use Automerge so that the reader gets an understanding of how Automerge works.

3.3.1 Map Object

We will start with the map object, which is similar to a standard JavaScript object. That is a dictionary with a text key to any object type as a value. We show an example below.

```
doc1 = Automerge.from(
    { country: "Norway", knownFor: "oil & gas" }
);
doc1 = Automerge.change(doc1, doc => {
    doc.knownFor = "fjords"
});
// doc1 = { country: "Norway", knownFor: "fjords" }
```

Listing 3.1: Use of map object

In the listing 3.1, we can see that we initialize the document with a JavaScript object using the `Automerge.from` function, and then we update the `knownFor` property using the `Automerge.change` function.

3.3.2 Merging Changes

A process running Automerge will collaborate with other processes, and they need to merge their changes eventually. A node can share its latest changes with another node by sending them across the network to another node. The other node can then apply the changes with the `Automerge.applyChanges` function. We show an example below where we continue from listing 3.1 and send the changes to another node.

```
changes = Automerge.getHistory(doc1);
network.broadcast(JSON.stringify(changes));

// On the other node
doc2 = Automerge.init();
changes = JSON.parse(network.receive());
doc2 = Automerge.applyChanges(doc2, changes);
```

Listing 3.2: Sending changes across network

In listing 3.2, we first get all the changes from the first node before we send those changes over to the second node. The receiving node gets the changes and applies them to an empty document using `Automerge.applyChanges`. The two nodes are then up to date with each other. If the second node had also made some changes, we would send them to the first node similarly.

If we have two documents in the same process, which is typical for testing, then it is simpler to merge changes using the `Automerge.merge` function, which we will use in most examples. In listing 3.3 below, we show how we can merge the changes of one document to the other, achieving the same as in listing 3.2.

```
doc2 = Automerge.init();
doc2 = Automerge.merge(doc2, doc1);
```

Listing 3.3: Merging two documents in same process

In listing 3.3 above, we merge the changes from doc1 into doc2. We assume doc1 has already made some changes similar to listing 3.1.

3.4 Other Types of Objects

In addition to the map object, which we will explore more in this thesis, Automerge provides several other types of objects that we will show in this section. Among these objects is the counter object for numbers that several nodes can increment. The counter object adds concurrent increments by several nodes, while if we used a JavaScript number in an object, then Automerge would choose only one of the concurrent increments.

For handling text, Automerge provides the text object which will merge concurrent changes to a collaborative text document. Automerge also provides the table object for handling tables similar to those found in databases that support referencing other tables and joining them together.

We show an example of the counter and text object below.

```
doc1 = Automerge.from({
    counter: new Automerge.Counter(),
    text: new Automerge.Text()
});
doc2 = Automerge.merge(Automerge.init(), doc1);
doc1 = Automerge.change(doc1, doc => {
    doc.counter.increment()
    doc.text.insertAt(0, 'foo')
});
doc2 = Automerge.change(doc2, doc => {
    doc.counter.increment()
    doc.text.insertAt(0, 'bar')
});
doc1 = Automerge.merge(doc1, doc2);
// doc1 could now be either
// { counter: 2, text = 'foobar' } or
// { counter: 2, text = 'barfoo' }
```

Listing 3.4: Counter and text object

Listing 3.4 shows two documents that increment a counter and add text at the same index in a text object. The counter increments get added, and the text gets merged when we merge the two documents. Note that since both documents insert text at the same index, Automerge will choose priority based on `ActorId`, which makes sure that all nodes that merge text will receive the same result.

3.5 Undo And Redo

Automerge supports undo and redo on local operations. A node can only undo the operations that it has created itself, and it cannot undo operations that originate from other nodes. In addition, it can only undo the last operation in the history. In order to support this, Automerge has an `undoStack`. Every time a node performs a local operation, it pushes the inverse operation onto the `undoStack`. Automerge also has a corresponding `redoStack`, where every time a node undoes an operation, it also pushes the inverse of that operation to the `redoStack`. Each stack has a stack pointer that keeps track of which operations have currently been undone and redone.

This implementation is limited in that it can only undo and redo the last local operation on a node. It also requires two stacks that both grow large when the number of operations made is substantial. In the most recent release of Automerge, the undo/redo feature is disabled, citing that "It was a bit of a hack" and that they hope to bring a better implementation in the future [7]. In the next chapter, we will show our solution to a better undo/redo implementation.

3.6 Architecture

The Automerge architecture consists of two main parts, the frontend, and the backend. Despite the naming, both parts of the architecture will run in the same process. However, it is possible to split the frontend and the backend apart and run them on different servers; this requires minor changes to the source code. Splitting the frontend and backend can be helpful in order to run the frontend and backend on different threads, such as offloading the UI thread. Note that Automerge does not provide any network functionality for communicating between nodes. The user will have to convert changes to JSON and send that to other nodes using the network functionality that they have available.

3.6.1 Frontend

Automerge presents the frontend as a JavaScript object to the user, which the user can modify with the `Automerge.change` function. This object also has several hidden properties created using JavaScript symbols [8, p. 41]. When the user creates a change locally, the frontend will detect the modifications made to the frontend object, and it will then create operations from these modifications and put them into a change sent to the backend. If a node receives a change from another node, the frontend will receive a patch from the backend instead of creating it.

3.6.2 Backend

The backend consists of an operation set [9] which is essentially a list of changes called the history, along with a queue. The operation set also contains other properties for keeping track of which dependencies the node has to on changes from other nodes.

A node handles changes from other nodes in the backend. The backend adds the changes to its queue, where they remain until the node applies the dependencies of the change. When all dependencies are satisfied, the change is applied and added to the history list. Then the backend sends a patch of the change to the frontend. When a node creates local changes, they are first created in the frontend, and then it sends a change request to the backend, where it applies the change to the backend.

4 Selective Undo and Redo of a Register

This chapter explains the abstract algorithm we use for selective undo and redo of a register using DAG ordering on operations. We do not discuss undo and redo on operations that edit different registers, as we show in the next chapter that this case is trivial. Instead, we only focus on changes to the same register.

4.1 Requirements

The current system for undo and redo in Automerge can only undo and redo the latest local change. Our system needs to undo and redo both local changes and changes from other nodes, and we should be able to undo/redo all previous changes known to our node. We list all the requirements for the system below and note that our examples only use a single value for our registers for simplicity. The result is still the same as values with different keys are primarily independent.

Our system needs concurrent updates between nodes and structural equality between them. We also need selective undo/redo, which means that undo and redo will be parameterized by the operation to be undone or redone. In figure

4.1, we can see that structural equality on operations means that two nodes can apply the same update for the same key, communicate that update, and when a node undoes that update, then the key is deleted. Without structural equality, node A would have two operations setting the value to 10, and undoing one of them would leave the other remaining.

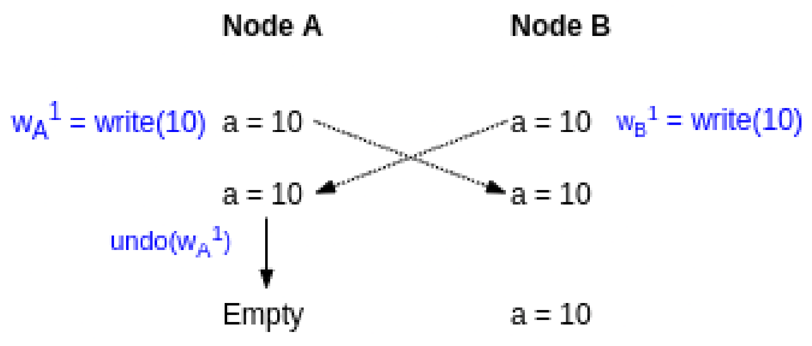

Figure 4.1: Structural Equality

Our system also needs a default resolution mechanism when two nodes make different concurrent changes.

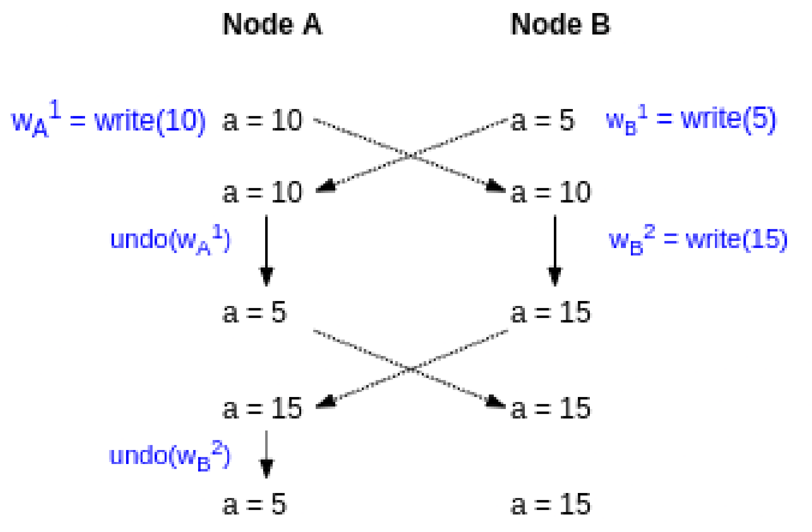

Figure 4.2: Conflict Resolution

In figure 4.2 we can see that both nodes resolve to the same value after communicating with each other. They do this with a deterministic resolution method. In Automerge, this resolution method consists of comparing the `actorIds` of the two nodes and giving priority to the node with the highest id. Figure 4.2 also shows that new updates should be prioritized over an undo/redo of an older update. We also expect every undo to reveal the most recent update, which we have not undone yet.

4.2 Undo-lengths

The undo-length for a normal operation is always zero, and it is then incremented by one whenever an operation is undone or redone. We can only undo an operation with an undo-length that is an even number, and we can only redo an operation if the undo-length is an odd number.

4.3 DAG ordering on Operations

Two operations from different nodes can be either concurrent or casually dependent on one another. If they are concurrent, we use a deterministic way of checking the priority of each operation that all nodes can calculate independently. This ordering allows us to create a partial order on normal operations, which we can visualize as a DAG that moves from the newest operation to the oldest. We show an example DAG below from figure 4.2. It shows the state of node A at the end. We show undo operations as inverse operations.

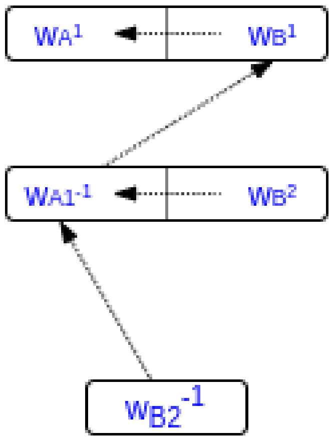

Figure 4.3: DAG ordering on operations

In figure 4.3 we can see the structure of the DAG. Concurrent operations share the same box, and we can think of the arrows between operations as signifying the "previous" operation. It tells us which operation we should make current in case we undo/redo the current operation. The current operation in figure 4.3 is the one on the bottom. Structurally, it is equal to the operation that it reinstated when it undid the previous operation. Note that when we apply deterministic conflict resolution to the concurrent operations, the DAG becomes a total order, which is essential to realize since the ordering needs to be total and deterministic.

When we undo an operation, we can use the DAG to access the previous operation to restore it. We restore the operation by creating another instance of it, adding an edge to the original operation.

5 Implementation

This chapter will explain the changes we have made to Automerge in order to support generic undo. We have chosen to focus on register changes in Automerge when implementing generic undo, which means we can set properties, update them, and delete them. Thus, we will not discuss undo on list, text, table, and counter objects. However, the method we use may apply to those objects as well. We would need to create inverse operations for all operations that those objects bring along.

We start by looking at some changes to the interface of Automerge before we discuss our solution using DAG ordering on Automerge changes.

5.1 Structural Equality on Changes

One main difference between the delta state-based CRDT used in previous implementations of undo-lengths [5] and that of Automerge is that changes that are equal in the structure are considered equal in delta state-based CRDTs even if they originate at different nodes. Automerge does not use structural equality; instead, it identifies a change by its sequence number and the actor id from the node from which it originates.

Structural equality on operations is essential when implementing generic undo through undo-lengths, as when two nodes undo the same change, the two undo operations have to be considered equal. It is for this reason that we implement structural equality on operations for Automerge. The change we made to Automerge is that when a node receives a change from another node, it will first check if it already has a structurally equal change to the change it receives. If it has such a change, then the node does not apply the incoming change.

With structural equality, we also add a new property to the Automerge backend. We call this property `changes` and it is a map from the hash of a change to the change itself. This is useful when implementing the DAG that we explain later.

5.2 Selective Undo

As we have mentioned in the undo section of the technical background chapter, there are two main ways of undoing edits. A common way is to allow the user to undo the latest edit and only redo the latest undo if the last edit was an undo. Automerge uses this approach; however, the undo-state CRDT [5] is a delta state-based CRDT that allows for selective undo.

Selective undo means that we can undo any edit, even if that edit is not the latest. With selective undo, we also have a selective redo, which is that we can redo any previously undone edit. To achieve selective undo, we have changed the undo and redo interface of Automerge, to instead use `Automerge.undoChange` and `Automerge.redoChange`. We show basic usage of this new interface below.

```
doc = Automerge.change(Automerge.init(), doc => { 'a': 10 });
changes = Automerge.getHistory(doc);
doc = Automerge.undoChange(doc, changes[0]);
changes = Automerge.getHistory(doc);
doc = Automerge.undoChange(doc, changes[1]);
```

Listing 5.1: Selective undo/redo interface

In listing 5.1, we first set a property `'a'` to the value 10 on an empty document. Then we get all the changes in the history of our document, which is just one in this case. We send this first change into `Automerge.undoChange`, to undo it, and then we proceed to redo the undo change afterward, thus restoring the property `'a'`.

5.3 DAG Ordering on Changes

As mentioned earlier, we will focus on register changes when nodes set properties on `Automerge.Map` objects. First, we consider the case where two nodes change different properties on the same map object, and they do not overwrite any properties. Changes on different nodes are the simplest case.

When nodes in Automerge change different properties on the same object, we can undo those changes by applying the inverse change, which is to delete the property. When we recognize that we have this scenario, we compute the inverse change from the change that we are undoing, and then we increment the undo-length from the change we are undoing.

The more complex scenario arises when nodes overwrite the value of the same property. That is, they overwrite a register value. When we undo a change that overwrites a register value, we must restore the previous value of the register. In order to restore the previous value, we must find out what that value is. To achieve that, we add a link to the previous value when we overwrite it. A link, in this case, is a hash of a change. We can use this hash to retrieve the change from the `changes` map, which we explained in section 5.1.

When changes that overwrite the same property are non-concurrent, this process is simple, as we can recognize that we overwrite a register value, then hash the previous change and add the hash as a link.

When concurrent changes overwrite the same property, there are two cases. The first case is when the local change is determined to be the first change based on the Automerge priority. In this case, we can apply the incoming remote change and add a link from the remote change to the local change.

In the other case where the local change is determined to be logically second to the remote change, we have to add a link from the local change to the remote change, and we must transfer any link that the local change might have over to the remote change.

5.4 Undo-lengths

We have discussed undo-length in the technical background. We set an undo-length on each Automerge change. It starts at zero, and Automerge increments the undo-length each time an operation is undone or redone. We use undo-length to prioritize between undo/redo changes of the equivalent original change, where the change with the highest undo-length has the highest priority.

If a node receives a change from another node with a lower undo-length than it currently has, then that change is ignored. In the other case, when the undo-length is higher, then the incoming change is applied and prioritized over the local change.

5.5 Undo and Redo Examples

In this section we show a detailed example of using the new selective undo/redo system, and how it uses DAG ordering and undo-lengths. For the following example, we assume that doc2 will have priority over doc1, which means, that `doc1.a` will be equal to 20 after the first merge.

```
doc1 = Automerge.from({'a': 10});
doc2 = Automerge.from({'a': 10});

doc1 = Automerge.change(doc1, doc => doc.a = 5);
doc2 = Automerge.change(doc2, doc => doc.a = 20);
doc1 = Automerge.merge(doc1, doc2);

changes = Automerge.history(doc1);
doc1 = Automerge.undoChange(doc1, changes[2]);

changes = Automerge.history(doc1);
doc1 = Automerge.redoChange(doc1, changes[3]);
```

Listing 5.2: Example of undo and redo

Figure 5.1 shows how the DAG is constructed in the history of doc1 at end of listing 5.2. From the example, we can see that undo and redo operations are applied as normal operations and contain all the necessary information as would a normal operation. However, they also have an undo-length showing how long their undo/redo chain is.

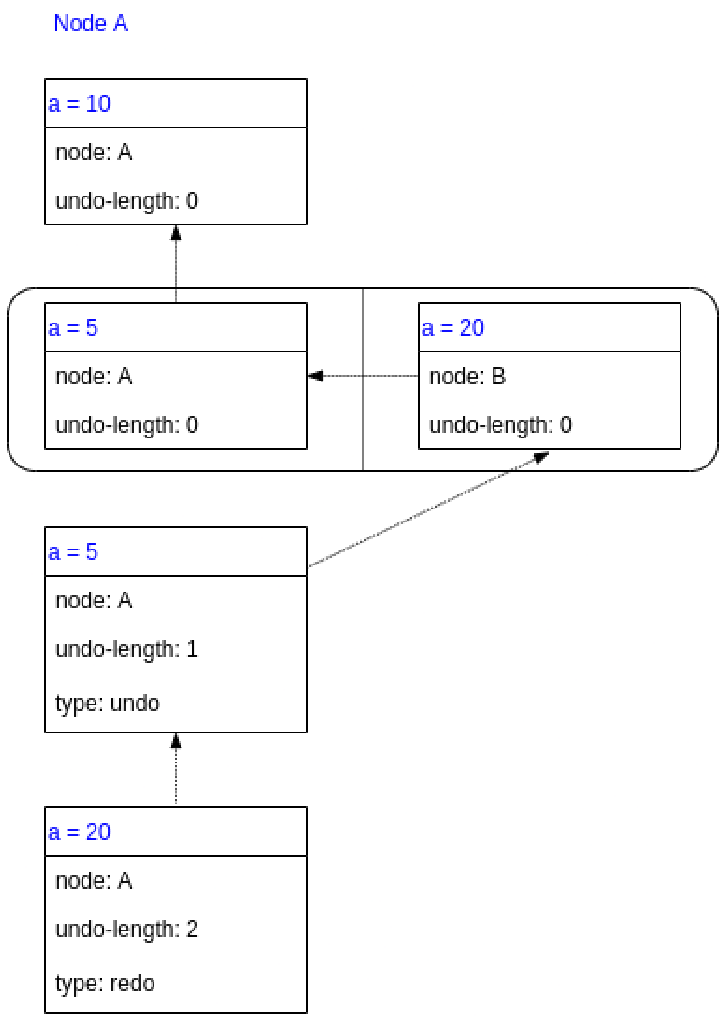

Figure 5.1: DAG in history of doc1

6 Discussion and Future Work

Automerge currently has limited undo and redo support. We have even seen that undo and redo support has been completely removed since its author deemed it unsatisfactory. In this thesis, we have designed a new way of implementing generic undo and redo for Automerge. There are still improvements to be made, which we discuss in the next section.

6.1 Future work

For this thesis, we have focused on allowing for generic undo on register operations in Automerge. However, Automerge has several data types in addition to registers.

6.1.1 Supporting Other Object Types

We have seen in chapter 3 that Automerge supports lists, counters, tables, and text. Even though we have only added support for register changes in Automerge, we believe that it is possible to support all other object types in Automerge using the same approach.

6.1.2 Conflict Resolution

We have seen that Automerge resolves conflict resolution by comparing which document has the highest actor-id. An improvement could be to let the user control the conflict resolution through an API, either manually or through a different algorithm. Greater control would allow the user to make better applications where conflicts are better suited to the application, decreasing the usage of undo and redo.

7 Conclusion

In this thesis, we have designed and implemented generic undo and redo support for Automerge, allowing users to undo any operation and not only local operations. In addition, we support selective undo and redo. There is still room for extending this work to support all data types in Automerge fully.

Bibliography

[1] D. Davey and H. Priestly, *Introduction to Order and Lattices*, 2nd ed. Cambridge University Press, 2002.

[2] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” in *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, ser. SSS’11, Grenoble, France: Springer-Verlag, 2011, pp. 386–400, ISBN: 9783642245497.

[3] P. S. Almeida, A. Shoker, and C. Baquero, “Delta state replicated data types,” *Journal of Parallel and Distributed Computing*, vol. 111, pp. 162–173, 2018.

[4] G. D. Abowd and A. J. Dix, “Giving undo attention,” *Interact. Comput.*, vol. 4, no. 3, pp. 317–342, 1992. DOI: `10.1016/0953-5438(92)90021-7`. [Online]. Available: `https://doi.org/10.1016/0953-5438(92)90021-7`.

[5] W. Yu, V. Elvinger, and C.-L. Ignat, “A generic undo support for state-based crdts,” in *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, P. Felber, R. Friedman, S. Gilbert, and A. Miller, Eds., ser. LIPIcs, vol. 153, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 14:1–14:17. DOI: `10.4230/LIPIcs.OPODIS.2019.14`. [Online]. Available: `https://doi.org/10.4230/LIPIcs.OPODIS.2019.14`.

[6] M. Kleppmann and A. R. Beresford, “A conflict-free replicated JSON datatype,” *CoRR*, vol. abs/1608.03960, 2016. arXiv: `1608.03960`. [Online]. Available: `http://arxiv.org/abs/1608.03960`.

[7] M. Kleppman, “Changelog,” 2021. [Online]. Available: `https://github.com/automerge/automerge/blob/main/CHANGELOG.md`.

[8] D. Flanagan, *JavaScript: The Definitive Guide*, 7th ed. O’Reilly Media, Inc, 2020.

[9] M. Kleppmann, V. B. Gomes, D. P. Mulligan, and A. R. Beresford, “Opsets: Sequential specifications for replicated datatypes (extended version),” *arXiv preprint arXiv:1805.04263*, 2018.

Appendix

For this thesis, my supervisor Weihai and I published a paper which has been accepted to the journal CDVE2021. The paper explains our algorithm for generic undo on registers. We show the paper below.

Supporting Undo and Redo for Replicated Registers in Collaborative Applications

Authors

Institutes
`emails`

Abstract. A collaborative application supporting eventual consistency may temporarily violate global invariant. Users may make mistakes. Undo and redo are a generic tool to restore global invariant and correct mistakes. A replicated register allows a collaborative application to concurrently read and write at different sites. Currently, there is very little undo and redo support of eventually consistent replicated registers. We present an approach to undo and redo support for eventually consistent replicated registers. We also present a work-in-progress implementation in a popular open-source library for collaborative applications.

Keywords: Data replication, optimistic concurrency control, eventual consistency, undo, redo

1 Introduction

Most collaborative applications replicate data at different sites and apply optimistic concurrency control that supports eventual consistency [12]. A system with eventual consistency may temporarily violate some global invariant, such as overbooking of resources. For applications such as online shopping and collaborative editing, human users introduce additional mistakes. Undo and redo are a generic tool to restore global invariant and correct human mistakes.

Register is one of the simplest and most fundamental data types. An application writes a value to a register and later read the value that it has written. For example, the font type of a document's title could be a register. An *eventually consistent replicated register*, or *EC register* for short, allows different sites to independently reads and writes their local register instances. The values of the instances converge when the sites have applied the same set of write updates to their local instances.

Undo and redo in collaborative applications are generally well understood and supported for immutable data elements that an application can insert into or delete from a composite data collection, such as a set or a document [2, 7, 11, 13, 15]. Undo and redo support for EC registers, however, has not been very well understood and supported.

In this paper, we first discuss the issues and requirements of undo and redo support for EC registers. We then present an approach to supporting undo and redo for EC registers. The approach is based on the causality of the write updates and their undo and redo updates. We also present a work-in-progress implementation of the approach in a popular open-source library for collaborative applications.

2 Technical issues

There exist two types of EC registers. LWW (last-write wins) register [5,8] is the mostly used EC register. Each write update of the register is associated with a timestamp (or priority in general). For two concurrent updates to the register, the one with the greatest timestamp value wins. The resolution to the conflict is thus lossy. The concurrent update that loses the competition gets lost.

Multi-value register [1, 9] makes lossless resolution among concurrent updates at the cost of application complexity. All concurrent updates are preserved and presented to the application. It is up to the application to decide a new value based on the multiple presented values.

When the updates to a register are sequential or serializable, such as in a single-user editor or an ACID (atomicity, consistency, isolation and durability) database, the system can maintain the history of the updates as a linear sequence of values. If the system knows the current position in the sequence, it performs an undo or redo by simply setting the register with the appropriate previous or next value in the sequence.

Under concurrent write updates, the update history is no longer linear. Unless we restrict what can be undone, for instance, by only allowing the undo or redo of the updates that were originated locally, finding a unique previous or next value is no longer trivial.

In addition to the normal write updates, undo and redo updates can also be performed concurrently at different sites. Neither do these undo and redo updates follow a linear order.

Essentially, we must address two issues. First, what is the current undo-redo status of the register? Second, given the current undo-redo status, what should be the appropriate value after the undo or redo?

Researchers count the number of undos and redos to figure out the current undo-redo status of an update [11,13,15]. We use undo length [15] for this purpose. For immutable values that are inserted into or deleted from a data collection, it is sometimes sufficient to perform an undo or redo when we are able to figure out the current undo-redo status. For EC registers, there has not been a solution for the second issue yet.

3 Requirements

An eventually consistent system must allow a site to independently perform updates to a register. The updates should include not only normal write updates but also undo and redo of any previously performed update.

When the sites are connected, they must be able to merge concurrent remote updates without any coordination. For example, they must be able to independently resolve conflicting updates without collecting votes from a quorum of sites.

The state of the register instances at different sites must be convergent. That is, when the sites have applied the same set of updates, regardless of the order in which the updates are applied, the instances must report the same register value.

The behavior of the register should be the same as a sequential system when the updates are sequential. As a special case, when we only make undo and redo on locally

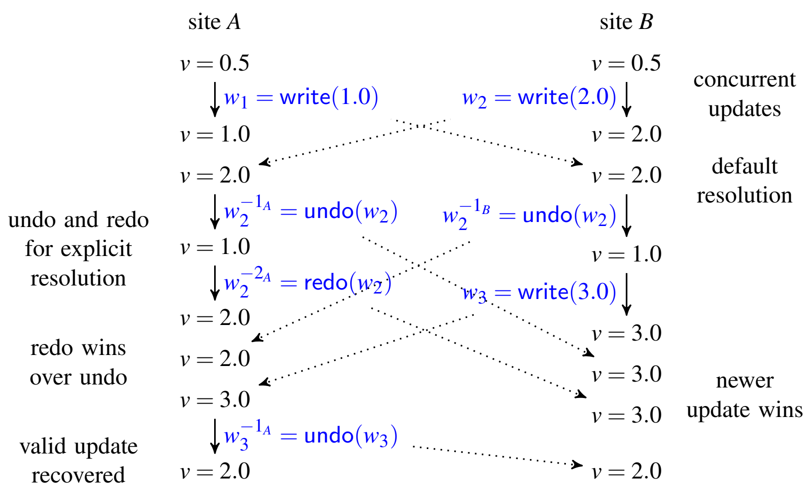

Fig. 1. A scenario of concurrent register updates

originated updates, the behavior should be the same as existing systems that only allow undo and redo of local-only updates.

The system should combine the benefits of current best practice, i.e. LWW and multi-value registers. On the one hand, the system should be able to resolves conflict among concurrent updates, with existing commonly applied conventions, such as LWW. On the other hand, while the system resolves conflict on behalf of the application or user, the application or user should still be able to make an explicit choice.

We illustrate the requirements with an example scenario. In Fig. 1, there are two instances of the same register at sites A and B. The instances have the same initial value 0.5. The sites independently write a new value with write updates w_1 and w_2. Suppose the priority (such as a timestamp value) of w_2 is greater than the priority of w_1. When receiving the update from the remote site, each site independently resolves the conflict and the update w_2 wins. The register instances at both sites have the same value 2.0.

Now, suppose the application at the two sites do not agree with the automatic resolution that the system has made, they can independently make an explicit resolution by undoing the update of w_2 with $w_2^{-1_A}$ and $w_2^{-1_B}$. The new value of the register now becomes 1.0.

Of course, the system must allow any undone update to be redone. If the application at site A regrets the undo and redoes w_2 with $w_2^{-2_A}$, the register at the site is restored back to value 2.0. Moreover, the redo $w_2^{-2_A}$ wins over the concurrent undo $w_2^{-1_B}$ (even though $w_2^{-1_B}$ arrives at site A after $w_2^{-2_A}$ has finished), because $w_2^{-1_B}$ has the same intention as $w_2^{-1_A}$ and site A had already seen the intention of $w_2^{-1_A}$ when it performed $w_2^{-2_A}$. In other words, we say that site A has now figured out that the current undo-redo status of w_2 is “redone”, or the update of w_2 is “effective”.

Now, site B performs a new write update w_3. It is critical that the sites can independently resolve the conflict between the new update w_3 and the concurrent undo and redo updates $w_2^{-1_A}$, $w_2^{-1_B}$ and $w_2^{-2_A}$. It makes sense that the latest update w_3 wins over the previous update w_2, including its undo and redo updates $w_2^{-1_A}$, $w_2^{-1_B}$ and $w_2^{-2_A}$. Hence the new value of the register becomes 3.0.

If site A now undoes w_3 with $w_3^{-1_A}$, the system restores the register with update w_2, which is currently redone. So the final value of the register becomes 2.0.

Note that if w_2 had been originated at site A, the behavior of the register would have been the same as a system that only allows undo and redo of locally originated updates.

4 Ordering normal, undo and redo updates

To resolve the conflicts of concurrent write updates and their undo and redo updates, the sites must be able to decide an order among them. In this section, we first consider the order of normal write updates and then take undo and redo into account.

4.1 Normal write updates

Two write updates are either concurrent or one of them is causally dependent on the other. For two write updates w and w', we write $w||w'$ if they are concurrent and $w \rightarrow w'$ if w and w' apply to the same register and w happens before w' (or w' causally depends on w).

The causality of updates on a register forms a partial order. We can draw a DAG (directed acyclic graph) where the vertices are the write updates on a register, and there is an edge from update w to update w' if $w \rightarrow w'$ and there does not exists an update w'' on the same register such that $w \rightarrow w''$ and $w'' \rightarrow w'$. That is, there is a direct (or immediate) causal dependency from w to w'.

Given a DAG of write updates on a register, we call an update w a *head* update in the DAG if there is no update w' in the DAG such that $w \rightarrow w'$. Clearly the head updates of the current DAG determine the current value of the register. The head updates are concurrent with each other and are originated from different sites.

A LWW register uses a priority to resolve the conflicts among the head updates, whereas a multi-value register present all head updates to the application which then makes a new update based on the presented updates. We propose that the system uses LWW to resolve the conflict, but an application can still explicitly choose a different concurrent update by undoing the current winning update.

4.2 Undo and redo updates

A commonly accepted semantic (or effect [10]) of an undo is that undoing an update w on a register has the same effect on the register where the update w has never occurred. This semantic should apply to concurrent undos of the same write update. When a write update w is undone, the causality of the still-effective updates that was established before the undo remains the same.

The redo of a write update w may have two alternative semantics. The first alternative is that w has never occurred (due to the undo) and a complete new update is applied. The second alternative is that the undo of w has never occurred and the effect of w is restored. We found the second alternative more natural and therefore choose that alternative. The effect of restoring w is that we have restored the DAG that was established before the undo of w.

Based on the discussion so far, we propose the following way to determine the order among the updates. We first build the DAG according to the causality of the updates. Undoing an update does not change the DAG. Instead, we mark the vertex for that undone update as ineffective. When the update is redone, we mark the vertex back to effective. The current value of the register is determined by the effective head updates.

A site usually only undoes the last effective update, since it does not make sense for an application to undo an update that is not currently in effect. The reader should not confuse this with selective undo [10, 13, 14] where an application can undo or redo any update in the history which contains the updates on all data objects (including different registers).

5 Undo lengths

Now that we are able to maintain the order among the updates on a register, the remaining task is to figure out whether a particular write update is effective at present. Here we adopt *undo length* [15] to figure out the current undo-redo status of a write update.

Notice that in Fig. 1, we used $w_2^{-1_A}$ and $w_2^{-2_A}$ to denote the undo and redo of update w_2 that site A performed. In general, we can use w^{l_s} to denote an undo or redo of update w that site s performs. Here the l in w^{l_s} (where $l > 0$) is called the undo length of w. w^{l_s} is an undo of w if l is an odd number. Otherwise, it is a redo. In other words, the undo-redo status of a write update is only dependent on its current undo length (i.e. it is independent of which sites that have performed the undo or redo updates).

6 High-level algorithms

For an EC register, a site maintains a set G of write updates. An update w is a 6-tuple $\langle o, k, p, l, v, D \rangle$, where o is the unique identifier of w, k is the vector clock [3] value, p is the priority, l is the undo length, v is the register value and D is the set of the write updates which w immediately depends on. For a write update w, we use k_w, p_w etc. to denote the k and p elements of w.

The set G represents the DAG of updates described in Section 4.1. Unlike traditional graph data structures, the links of the DAG are maintained backward, through the immediate causal dependencies (starting from the head updates). For a set G of write updates, we also maintain $H_G \subseteq G$, the head updates of G.

Initially, the set G is empty.

```
$w \leftarrow \langle \mathsf{newId}(), \mathsf{readClock}(G), \mathsf{getPriority}(), 0, v, H_G \rangle$
$G \leftarrow G \cup \{w\}$
$H_G \leftarrow \{w\}$
**return** $w$
```

Algorithm 1: Local update $\mathsf{write}(v)$

When writing a new value v to the register (Algorithm 1), we generate a globally unique identifier, a new vector clock value and a priority for the new write update w. For any write update w' in G, the new vector clock value $k_w > k_{w'}$. Furthermore, for any write update w'' that is currently not in G, $k_w \not> k_{w''}$. The new update w depends immediately on the head updates in H_G. The initial undo length of w is 0. We insert w into G. The new head of G consists only of this new write update w.

```
$w \leftarrow G.\mathsf{find}(o)$
if $w \wedge \mathsf{even}(l_w)$ then
    $l_w \leftarrow l_w + 1$
    return $\langle o, l_w \rangle$
```

Algorithm 2: Local undo $\mathsf{undo}(o)$

We can only undo an effective update. An update is effective when its undo length is an even number. To undo an effective update, we simply increment its undo length with 1 (Algorithm 2).

```
$w \leftarrow G.\mathsf{find}(o)$
if $w \wedge \mathsf{odd}(l_w)$ then
    $l_w \leftarrow l_w + 1$
    return $\langle o, l_w \rangle$
```

Algorithm 3: Local redo $\mathsf{redo}(o)$

Similarly, we can only redo an ineffective update, whose undo length is an odd number. To redo the update, we simply increment its undo length with 1 (Algorithm 3).

A site broadcasts the representation of local updates returned by Algorithms 1, 2 and 3 to remote sites.

A site merges an incoming remote update only when the update is causally ready, i.e. when the site has applied all the updates which the incoming update depends on.

```
$G \leftarrow G \cup \{w\}$
$H' \leftarrow \{w' \in H_G | k_{w'} < k_w\}$
$H_G \leftarrow (H_G \setminus H') \cup \{w\}$
```

Algorithm 4: Merge update w

To merge a new write update w (Algorithm 4), we insert w into G. Since update w may have already seen some of the head updates of this site, we remove from H_G the updates that w has seen (with clock values less than k_w), and then add w as a new head update.

```
$w \leftarrow G.\mathsf{find}(o)$
$l_w \leftarrow \mathsf{max}(l_w, l)$
```

Algorithm 5: Merge undo or redo $\langle o, l \rangle$

To merge an undo or redo update (Algorithm 5), we update the undo length of the write update. The new undo length is the greater one of the incoming undo length l and the undo length l_w that has been locally recorded.

```
$H \leftarrow H_G$
while $H \neq \emptyset$ do
    $H_e \leftarrow \{w \in H \,|\, \mathsf{even}(l_w)\}$
    if $H_e \neq \emptyset$ then
        return $\mathsf{resolve}(H_e)$
    $D_H \leftarrow \bigcup_{w \in H} D_w$
    $D_{D_H} \leftarrow \{w \in D_H \,|\, \exists w' \in D_H : k_w < k_{w'}\}$
    $H \leftarrow D_H \setminus D_{D_H}$
return UNDEFINED
```

Algorithm 6: Query current value *read*()

To get the current value of the register (Algorithm 6), we must obtain the head updates of the current effective sub-graph of G. We do this in a loop that starts with the head updates of G. In the loop, H is the set of updates that are the current candidates of effective head updates. We first get the effective updates $H_e \subseteq H$, the updates whose undo lengths are even numbers. If H_e is not empty, we get the register value with the resolve function which resolves the conflicts among the effective updates in H_e using the priorities of the updates. That is, the resolve function returns value v_w of update w in H_e such that for all w' in H_e, $p_w \geq p_{w'}$.

If none of the current head updates in H is effective, we try to obtains a set of new head updates from the sub-DAG $(G \setminus H)$. To get the new head updates, we first get D_H, the set of updates that the updates in H depend immediately on. We then eliminate the updates in D_H that some other updates in D_H depends on.

We iterate over the sets of head updates of the sub-graphs until we get an effective write update. If no write update in G is effective, the query returns a special UNDEFINED value.

7 A work-in-progress implementation

In Section 6 we presented the algorithms for EC registers at a rather high and abstract level. In this section, we report our work-in-progress implementation in Automerge[1], a popular open source library for collaborative applications. Briefly, Automerge is a Javascript library of a JSON CRDT [6]. A CRDT (conflict-free replicated data type) [9] is a data abstraction specifically designed for data replicated at different sites. The sites can independently query and update the local CRDT instances. A CRDT guarantees that when all sites have applied the same set of updates, the states of the instances at these sites converge.

A JSON[2] document is a tree of nodes. A branching (or intermediate) node is either a key-value map (also known as an "object" or "name-value pairs") or a sequential list. In a map, a key is, at any given time, associated with a value. A key in a map is basically a register. In what follows, we will restrict our discussions on the setting (i.e. writing) and fetching (i.e. reading) the value of a given key.

At present, Automerge only allows a site to undo and redo the latest updates that are originated locally at the site. To achieve this, a site maintains an undo stack and a redo

[1] `https://github.com/automerge/automerge`
[2] `https://www.json.org/json-en.html`

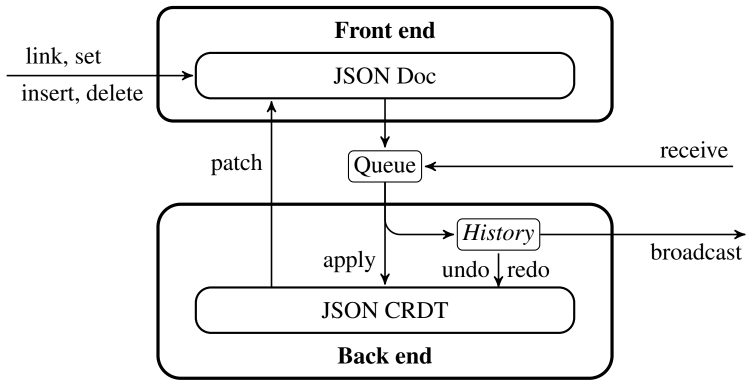

Fig. 2. Structure of a site

stack. When the site performs a local update, it generates a reverse update and pushes it to the undo stack. To perform an undo, it pops and performs an update from the undo stack, and also pushes the corresponding original update into the redo stack.

In our current implementation, we have removed the undo and redo stacks from Automerge. Fig. 2 shows the revised software structure of a site. Instead of using the undo and redo stacks, we now maintain the information necessary for undo and redo in the *history* of updates.

A site consists of a front end and a back end (Fig. 2). A collaborative application updates and queries the JSON document in the front end. The update operations include linking a new node to the JSON tree (including linking a new key to a key-value map), setting a new value to a key in a map, inserting a data element into a list and deleting an element from the list.

The JSON CRDT is stored in the back end. When the front end has performed a local update, it inserts an update for the CRDT in the queue. The queue also contains the remote updates the site has received.

The back end performs the updates in the queue that are causally ready. After it has performed a remote update, it generates an update (called a patch) for the front end. Basically, we can regard the front-end document as the cache of the query result on the latest back-end CRDT state.

The back end also maintains a history of updates that it has performed. One purpose of the history is for the site to synchronize its local updates with remote sites. Now, we also use the history for the purpose of undo and redo.

The representation of an Automerge update (an *update object*) for the JSON CRDT contains some meta data, including the unique identifier of the site (known as actor-id) that originally performed the update, the sequence number of the update at the original site, the updates that this update immediately depends on, the unique identifier of the data element, etc. An update is uniquely identified with the (actor-id, sequence number) pair. Automerge uses actor-ids as priorities to resolve conflicts between concurrent write updates.

1. Site A after $w_2^{-2_A}$

$$\begin{tikzcd}
w_1 & w_2 \arrow[l] & w_2^{-1_A} \arrow[l] & w_2^{-2_A} \arrow[l] \\
 & & & \text{head} \arrow[u]
\end{tikzcd}$$

2. Site A after $w_2^{-1_B}$

$$\begin{tikzcd}
w_1 & w_2 \arrow[l] & w_2^{-1_A} \arrow[l] & w_2^{-2_A} \arrow[r] & w_2^{-1_B} \arrow[ll, bend right] \\
 & & & \text{head} \arrow[u] &
\end{tikzcd}$$

3. Site A final

$$\begin{tikzcd}
w_1 & w_2 \arrow[l] & w_2^{-1_A} \arrow[l] & w_2^{-2_A} \arrow[r] & w_2^{-1_B} \arrow[ll, bend right] & w_3 \arrow[ll, bend left] & w_3^{-1_A} \arrow[l] \\
 & & & & & & \text{head} \arrow[u]
\end{tikzcd}$$

4. Site B final

$$\begin{tikzcd}
w_2 \arrow[r] & w_1 & w_2^{-1_B} \arrow[ll, bend right] & w_3 \arrow[rr, bend left] & w_2^{-1_A} \arrow[ll, bend left] & w_2^{-2_A} \arrow[l] & w_3^{-1_A} \arrow[lll, bend right] \\
 & & & & & & \text{head} \arrow[u]
\end{tikzcd}$$

Fig. 3. Histories of updates

In order to implement the high-level algorithms presented in Section 6, we augmented the update objects with additional meta data. More specifically, an update object in the history contains now an additional link to a previous update object as well as the undo length of the update. The update objects of a register are now linked sequentially in a total order that respects both the partial order of the DAG (Section 4.1) and the priorities of the concurrent updates. In other words, the links now encode different types of relations between updates, namely, the immediate causal dependencies, priority order, and undo and redo updates. A site only maintains the links locally and does not include them when it sends update objects to remote sites.

Fig. 3 shows the update histories at the two sites for the scenario illustrated in Fig. 1. When the back end has performed an update, it appends an update object to the end of the history. If the register happens to be updated in causal order or in the order in which the conflicts are resolved (Fig. 3-1), we link the new update object to the last update object of the register in the history. If a new remote update loses the competition with a concurrent update (Fig. 3-2), we re-arrange the links so that the linked list respects the order for conflict resolution. Fig. 3-3 and Fig. 3-4 show the final histories at sites A and B.

To perform a remote update, we first check if the register exists using the identifier of the updated data element. If it does not exist, we simply perform the update, append the update object to the end of the history and mark this object as the head update. If the register exists, we append the update object to the history and scan the history backward until we find the head update of the register. We re-arrange the links and set the new head update using the meta data in the update objects. Then, we start from the new head update and follow the links until we reach an effective update. We generate a patch according to that effective update and send patch to the front end. While traversing through the links, when we have visited an ineffective update, we skip over all the objects of that update. For example, in Fig. 3-3, when we have visited $w_3^{-1_A}$, we skip w_3 and visit $w_2^{-2_A}$, which is effective. So we generate a patch that writes the register with value 2.0. Notice that because $w_2^{-1_A}$ and $w_2^{-1_B}$ have the same effectiveness, their order in the sequential list does not play any role. Therefore the final histories at sites A and B are in fact equivalent.

In summary, the update history maintains two different orders. The physical order in the history allows the different sites to synchronize with each when they get connected. The linked list of the update objects of the same register allows the sites to independently determine the current value of the register.

8 Related work

Supporting undo and redo for concurrent updates has been an active research topic, both in the area of collaborative editing [10, 11, 13, 14] and CRDTs for general-purpose collaborative applications [15]. At present, there is no general support for undo and redo of concurrent updates on eventually consistent replicated registers.

The authors of [15] presented an approach to generic undo support for CRDTs. In [15], an update is represented with a join-irreducible state of the (state-based) CRDT where the states form a join-semilattice [4]. Defining appropriate join-irreducible states for replicated registers is non-trivial. For example, LWW registers [5, 8] use timestamp values as the order of the join-semilattice, but timestamp values may not sufficiently capture the causality of the updates. For multi-value registers [1, 9], it is not clear what a previous value should be when we apply an undo to a write update. We address the issues by combining different orders on the updates: the causal order of the normal write updates (similar to multi-value register), the priority order of concurrent write updates (similar to LWW register), and the order of undo and redo updates of a particular write update (via undo length [15]). We then presented the high-level algorithms for maintaining and using these different orders. Finally, we presented an implementation that combined these different orders into a single total order.

9 Conclusion

We have first discussed the issues and requirements of undo and redo support for eventually consistent replicated registers, and then presented a new approach. The new approach is based on the causality of the write updates and their undo and redo updates. The approach addresses the issues and meets the requirements, and it embodies the existing best practice of replicated registers. When the system automatically resolves conflicts among concurrent updates, the new approach falls back to LWW registers. The application using this approach is able to explicitly resolve conflicts through undo and redo, surpassing the capability of multi-value registers. When undo and redo are restricted to locally originated updates, the approach behaves the same as existing systems with such restriction. However, this new approach is not just an ensemble of the current best practice. It allows an application to undo and redo any update, which no existing system supports.

References

1. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)* (2007), pp. 205–220.
2. Ferrié, J., Vidot, N., and Cart, M. Concurrent undo operations in collaborative environments using operational transformation. In *CoopIS/DOA/ODBASE (1)* (2004), pp. 155–173.
3. Fidge, C. J. Logical time in distributed computing systems. *Computer 24*, 8 (1991), 28–33.
4. Garg, V. K. *Introduction to Lattice Theory with Computer Science Applications*. Wiley, 2015.
5. Johnson, P., and Thomas, R. The maintamance of duplicated databases. *Internet Request for Comments RFC 677* (January 1976).
6. Kleppmann, M., and Beresford, A. R. A conflict-free replicated JSON datatype. *IEEE Trans. Parallel Distrib. Syst. 28*, 10 (2017), 2733–2746.
7. Ressel, M., and Gunzenhäuser, R. Reducing the problems of group undo. In *GROUP* (1999), ACM, pp. 131–139.
8. Shapiro, M., Preguiça, N. M., Baquero, C., and Zawirski, M. A comprehensive study of convergent and commutative replicated data types. *Rapport de recherche 7506* (January 2011).
9. Shapiro, M., Preguiça, N. M., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, (SSS 2011)* (2011), pp. 386–400.
10. Sun, C. Undo as concurrent inverse in group editors. *ACM Trans. Comput.-Hum. Interact. 9*, 4 (2002), 309–361.
11. Sun, D., and Sun, C. Context-based operational transformation in distributed collaborative editing systems. *IEEE Trans. Parallel Distrib. Syst. 20*, 10 (2009), 1454–1470.
12. Vogels, W. Eventually consistent. *Comminications of the ACM 52*, 1 (2009), 40–44.
13. Weiss, S., Urso, P., and Molli, P. Logoot-undo: Distributed collaborative editing system on P2P networks. *IEEE Trans. Parallel Distrib. Syst. 21*, 8 (2010), 1162–1174.
14. Yu, W., André, L., and Ignat, C.-L. A CRDT supporting selective undo for collaborative text editing. In *DAIS* (2015), pp. 193–206.
15. Yu, W., Elvinger, V., and Ignat, C.-L. A generic undo support for state-based CRDTs. In *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)* (2020), vol. 153 of *LIPIcs*, pp. 14:1–14:17.