A Conflict-Free Replicated JSON Datatype

Martin Kleppmann and Alastair R. Beresford

Abstract—Many applications model their data in a general-purpose storage format such as JSON. This data structure is modified by the application as a result of user input. Such modifications are well understood if performed sequentially on a single copy of the data, but if the data is replicated and modified concurrently on multiple devices, it is unclear what the semantics should be. In this paper we present an algorithm and formal semantics for a JSON data structure that automatically resolves concurrent modifications such that no updates are lost, and such that all replicas converge towards the same state (a conflict-free replicated datatype or CRDT). It supports arbitrarily nested list and map types, which can be modified by insertion, deletion and assignment. The algorithm performs all merging client-side and does not depend on ordering guarantees from the network, making it suitable for deployment on mobile devices with poor network connectivity, in peer-to-peer networks, and in messaging systems with end-to-end encryption.

Index Terms—CRDTs, Collaborative Editing, P2P, JSON, Optimistic Replication, Operational Semantics, Eventual Consistency.

1 Introduction

Users of mobile devices, such as smartphones, expect applications to continue working while the device is offline or has poor network connectivity, and to synchronize its state with the user's other devices when the network is available. Examples of such applications include calendars, address books, note-taking tools, to-do lists, and password managers. Similarly, collaborative work often requires several people to simultaneously edit the same text document, spreadsheet, presentation, graphic, and other kinds of document, with each person's edits reflected on the other collaborators' copies of the document with minimal delay.

What these applications have in common is that the application state needs to be replicated to several devices, each of which may modify the state locally. The traditional approach to concurrency control, serializability, would cause the application to become unusable at times of poor network connectivity [1]. If we require that applications work regardless of network availability, we must assume that users can make arbitrary modifications concurrently on different devices, and that any resulting conflicts must be resolved.

The simplest way to resolve conflicts is to discard some modifications when a conflict occurs, for example using a "last writer wins" policy. However, this approach is undesirable as it incurs data loss. An alternative is to let the user manually resolve the conflict, which is tedious and error-prone, and therefore should be avoided whenever possible.

Current applications solve this problem with a range of ad-hoc and application-specific mechanisms. In this paper we present a general-purpose datatype that provides the full expressiveness of the JSON data model, and supports concurrent modifications without loss of information. As we shall see later, our approach typically supports the automatic merging of concurrent modifications into a JSON data structure. We introduce a single, general mechanism (a multi-value register) into our model to record conflicting updates to leaf nodes in the JSON data structure. This mechanism then provides a consistent basis on which applications can resolve any remaining conflicts through programmatic means, or via further user input. We expect that implementations of this datatype will drastically simplify the development of collaborative and state-synchronizing applications for mobile devices.

M. Kleppmann and A.R. Beresford are with the University of Cambridge Computer Laboratory, Cambridge, UK.
Email: mk428@cl.cam.ac.uk, arb33@cl.cam.ac.uk.

1.1 JSON Data Model

JSON is a popular general-purpose data encoding format, used in many databases and web services. It has similarities to XML, and we compare them in Section 3.2. The structure of a JSON document can optionally be constrained by a schema; however, for simplicity, this paper discusses only untyped JSON without an explicit schema.

A JSON document is a tree containing two types of branch node:

- **Map:** A node whose children have no defined order, and where each child is labelled with a string *key*. A key uniquely identifies one of the children. We treat keys as immutable, but values as mutable, and key-value mappings can be added and removed from the map. A JSON map is also known as an *object*.
- **List:** A node whose children have an order defined by the application. The list can be mutated by inserting or deleting list elements. A JSON list is also known as an *array*.

A child of a branch node can be either another branch node, or a leaf node. A leaf of the tree contains a primitive value (string, number, boolean, or null). We treat primitive values as immutable, but allow the value of a leaf node to be modified by treating it as a *register* that can be assigned a new value.

This model is sufficient to express the state of a wide range of applications. For example, a text document can be represented by a list of single-character strings; character-by-character edits are then expressed as insertions and deletions of list elements. In Section 3.1 we describe four more complex examples of using JSON to model application data.

1.2 Replication and Conflict Resolution

We consider systems in which a full copy of the JSON document is replicated on several devices. Those devices could be servers in datacenters, but we focus on mobile devices such as smartphones and laptops, which have intermittent network connectivity. We do not distinguish between devices owned by the same user and different users. Our model allows each device to optimistically modify its local replica of the document, and to asynchronously propagate those edits to other replicas.

Our only requirement of the network is that messages sent by one replica are eventually delivered to all other replicas, by retrying if delivery fails. We assume the network may arbitrarily delay, reorder and duplicate messages.

Our algorithm works client-side and does not depend on any server to transform or process messages. This approach allows messages to be delivered via a peer-to-peer network as well as a secure messaging protocol with end-to-end encryption [2]. The details of the network implementation and cryptographic protocols are outside of the scope of this paper.

In Section 4 we define formal semantics describing how conflicts are resolved when a JSON document is concurrently modified on different devices. Our design is based on three simple principles:

1) All replicas of the data structure should automatically converge towards the same state (a requirement known as *strong eventual consistency* [3]).
2) No user input should be lost due to concurrent modifications.
3) If all sequential permutations of a set of updates lead to the same state, then concurrent execution of those updates also leads to the same state [4].

1.3 Our Contributions

Our main contribution in this work is to define an algorithm and formal semantics for collaborative, concurrent editing of JSON data structures with automatic conflict resolution. Although similar algorithms have previously been defined for lists, maps and registers individually (see Section 2), to our knowledge this paper is the first to integrate all of these structures into an arbitrarily composable datatype that can be deployed on any network topology.

A key requirement of conflict resolution is that after any sequence of concurrent modifications, all replicas eventually converge towards the same state. In Section 4.4 and the appendix we prove a theorem to show that our algorithm satisfies this requirement.

Composing maps and lists into arbitrarily nested structures opens up subtle challenges that do not arise in flat structures, due to the possibility of concurrent edits at different levels of the tree. We illustrate some of those challenges by example in Section 3.1. Nested structures are an important requirement for many applications. Consequently, the long-term goal of our work is to simplify the development of applications that use optimistic replication by providing a general algorithm for conflict resolution whose details can largely be hidden inside an easy-to-use software library.

2 Related Work

In this section we discuss existing approaches to optimistic replication, collaborative editing and conflict resolution.

2.1 Operational Transformation

Algorithms based on *operational transformation* (OT) have long been used for collaborative editing applications [5], [6], [7], [8]. Most of them treat a document as a single ordered list (of characters, for example) and do not support nested tree structures that are required by many applications. Some algorithms generalize OT to editing XML documents [9], [10], [11], which provides nesting of ordered lists, but these algorithms do not support key-value maps as defined in this paper (see Section 3.2). The performance of OT algorithms degrades rapidly as the number of concurrent operations increases [12], [13].

Most deployed OT collaboration systems, including Google Docs [14], Etherpad [15], Novell Vibe [16] and Apache Wave (formerly Google Wave [11]), rely on a single server to decide on a total ordering of operations [17], a design decision inherited from the Jupiter system [8]. This approach has the advantage of making the transformation functions simpler and less error-prone [18], but it does not meet our requirements, since we want to support peer-to-peer collaboration without requiring a single server.

Many secure messaging protocols, which we plan to use for encrypted collaboration, do not guarantee that different recipients will see messages in the same order [2]. Although it is possible to decide on a total ordering of operations without using a single server by using an atomic broadcast protocol [19], such protocols are equivalent to consensus [20], so they can only safely make progress if a quorum of participants are reachable. We expect that in peer-to-peer systems of mobile devices participants will frequently be offline, and so any algorithm requiring atomic broadcast would struggle to reach a quorum and become unavailable. Without quorums, the strongest guarantee a system can give is causal ordering [21].

The Google Realtime API [22] is to our knowledge the only implementation of OT that supports arbitrary nesting of lists and maps. Like Google Docs, it relies on a single server [17]. As a proprietary product, details of its algorithms have not been published.

2.2 CRDTs

Conflict-free replicated datatypes (CRDTs) are a family of data structures that support concurrent modification and guarantee convergence of concurrent updates. They work by attaching additional metadata to the data structure, making modification operations commutative by construction. The JSON datatype described in this paper is a CRDT.

CRDTs for registers, counters, maps, and sets are well-known [3], [23], and have been implemented in various deployed systems such as Riak [24], [25]. For ordered lists, various algorithms have been proposed, including WOOT [26], RGA [27], Treedoc [28], Logoot [29], and LSEQ [30]. Attiya et al. [31] analyze the metadata overhead of collaboratively edited lists, and provide a correctness proof of the RGA algorithm. However, none of them support nesting: all of

the aforementioned algorithms assume that each of their elements is a primitive value, not another CRDT.

The problem of nesting one CRDT inside another (also known as *composition* or *embedding*) has only been studied more recently. Riak allows nesting of counters and registers inside maps, and of maps within other maps [24], [25]. Embedding counters inside maps raises questions of semantics, which have been studied by Baquero, Almeida and Lerche [32]. Almeida et al. [33] also define delta mutations for nested maps, and Baquero et al. [34] define a theoretical framework for composition of state-based CRDTs, based on lattices. None of this work integrates CRDTs for ordered lists, but the treatment of causality in these datatypes forms a basis for the semantics developed in this paper.

Burckhardt et al. [35] define *cloud types*, which are similar to CRDTs and can be composed. They define *cloud arrays*, which behave similarly to our map datatype, and *entities*, which are like unordered sets or relations; ordered lists are not defined in this framework.

On the other hand, Martin et al. [36] generalize Logoot [29] to support collaborative editing of XML documents – that is, a tree of nested ordered lists without nested maps. As discussed in Section 3.2, such a structure does not capture the expressiveness of JSON.

Although CRDTs for registers, maps and ordered lists have existed for years in isolation, we are not aware of any prior work that allows them all to be composed into an arbitrarily nested CRDT with a JSON-like structure.

2.3 Other Approaches

Many replicated data systems need to deal with the problem of concurrent, conflicting modifications, but the solutions are often ad-hoc. For example, in Dynamo [37] and CouchDB, if several values are concurrently written to the same key, the database preserves all of these values, and leaves conflict resolution to application code – in other words, the only datatype it supports is a multi-value register. Naively chosen merge functions often exhibit anomalies such as deleted items reappearing [37]. We believe that conflict resolution is not a simple matter that can reasonably be left to application programmers.

Another frequently-used approach to conflict resolution is *last writer wins* (LWW), which arbitrarily chooses one among several concurrent writes as "winner" and discards the others. LWW is used in Apache Cassandra, for example. It does not meet our requirements, since we want no user input to be lost due to concurrent modifications.

Resolving concurrent updates on tree structures has been studied in the context of file synchronization [38], [39].

Finally, systems such as Bayou [40] allow offline nodes to execute transactions tentatively, and confirm them when they are next online. This approach relies on all servers executing transactions in the same serial order, and deciding whether a transaction was successful depending on its preconditions. Bayou has the advantage of being able to express global invariants such as uniqueness constraints, which require serialization and cannot be expressed using CRDTs [41]. Bayou's downside is that tentative transactions may be rolled back, requiring explicit handling by the application, whereas CRDTs are defined such that operations cannot fail after they have been performed on one replica.

3 Composing Data Structures

In this section we informally introduce our approach to collaborative editing of JSON data structures, and illustrate some peculiarities of concurrent nested data structures. A formal presentation of the algorithm follows in Section 4.

3.1 Concurrent Editing Examples

The sequential semantics of editing a JSON data structure are well-known, and the semantics of concurrently editing a flat map or list data structure have been thoroughly explored in the literature (see Section 2). However, when defining a CRDT for JSON data, difficulties arise due to the interactions between concurrency and nested data structures.

In the following examples we show some situations that might occur when JSON documents are concurrently modified, demonstrate how they are handled by our algorithm, and explain the rationale for our design decisions. In all examples we assume two replicas, labelled p (drawn on the left-hand side) and q (right-hand side). Local state for a replica is drawn in boxes, and modifications to local state are shown with labelled solid arrows; time runs down the page. Since replicas only mutate local state, we make communication of state changes between replicas explicit in our model. Network communication is depicted with dashed arrows.

Our first example is shown in Figure 1. In a document that maps "key" to a register with value "A", replica p sets the value of the register to "B", while replica q concurrently sets it to "C". As the replicas subsequently exchange edits via network communication, they detect the conflict. Since we do not want to simply discard one of the edits, and the strings "B" and "C" cannot be meaningfully merged, the system must preserve both concurrent updates. This datatype is known as a *multi-value register*: although a replica can only assign a single value to the register, reading the register may return a set of multiple values that were concurrently written.

A multi-value register is hardly an impressive CRDT, since it does not actually perform any conflict resolution. We use it only for primitive values for which no automatic merge function is defined. Other CRDTs could be substituted in its place: for example, a counter CRDT for a number that can only be incremented and decremented, or an ordered list of characters for a collaboratively editable string (see also Figure 4).

Figure 2 gives an example of concurrent edits at different levels of the JSON tree. Here, replica p adds "red" to a map of colors, while replica q concurrently blanks out the entire map of colors and then adds "green". Instead of assigning an empty map, q could equivalently remove the entire key "colors" from the outer map, and then assign a new empty map to that key. The difficulty in this example is that the addition of "red" occurs at a lower level of the tree, while concurrently the removal of the map of colors occurs at a higher level of the tree.

One possible way of handling such a conflict would be to let edits at higher levels of the tree always override concurrent edits within that subtree. In this case, that would mean the addition of "red" would be discarded, since it

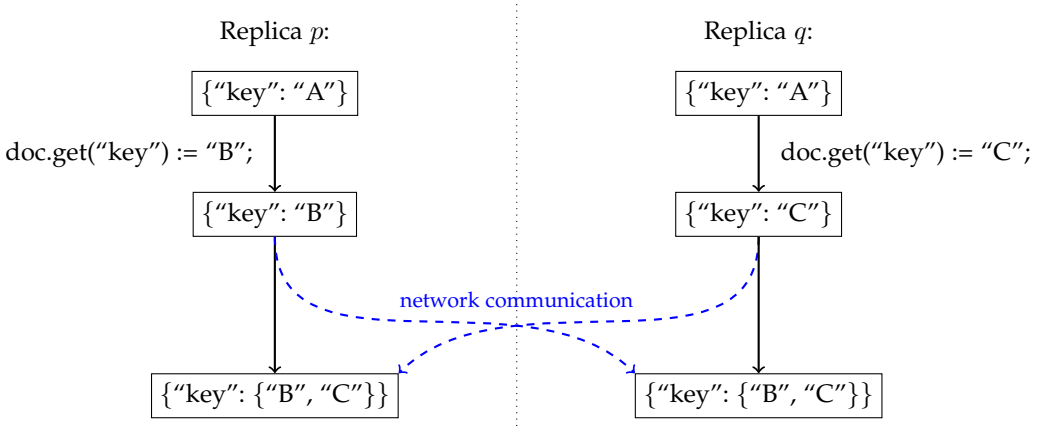

Fig. 1. Concurrent assignment to the register at doc.get(“key”) by replicas p and q.

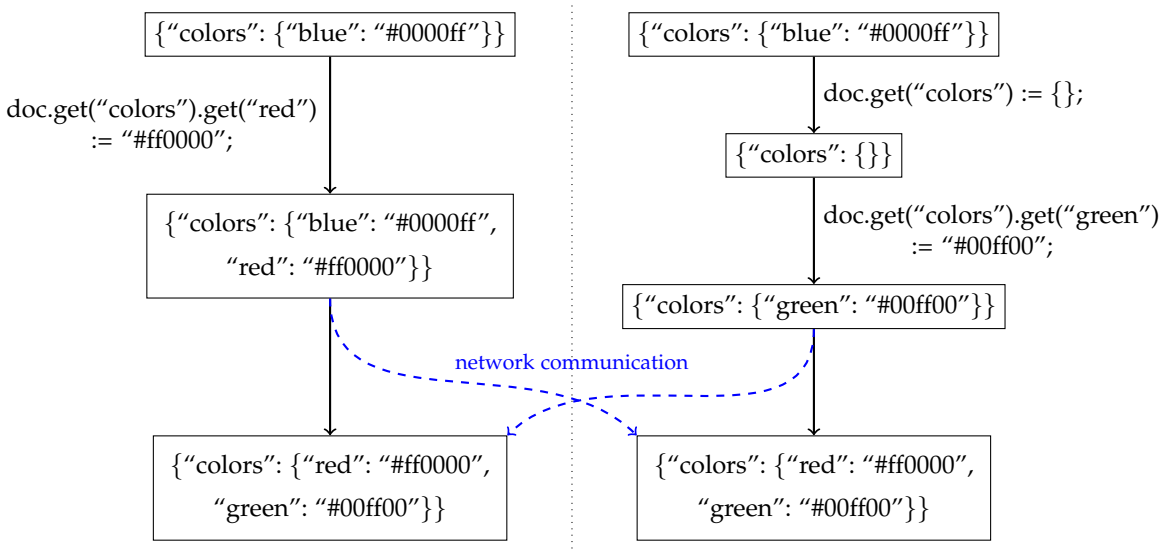

Fig. 2. Modifying the contents of a nested map while concurrently the entire map is overwritten.

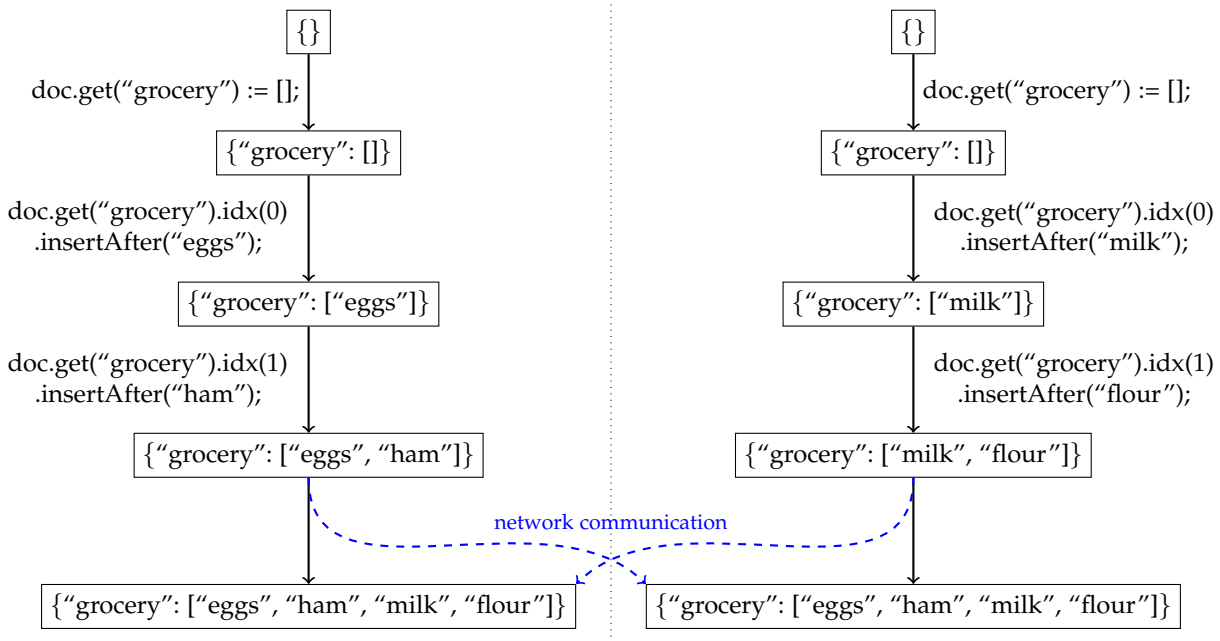

Fig. 3. Two replicas concurrently create ordered lists under the same map key.

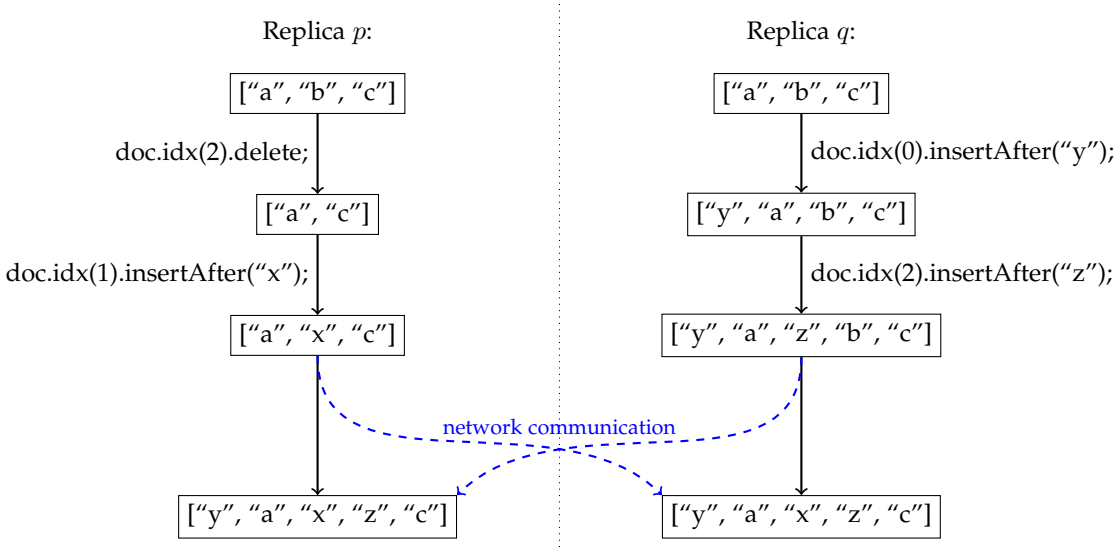

Fig. 4. Concurrent editing of an ordered list of characters (i.e., a text document).

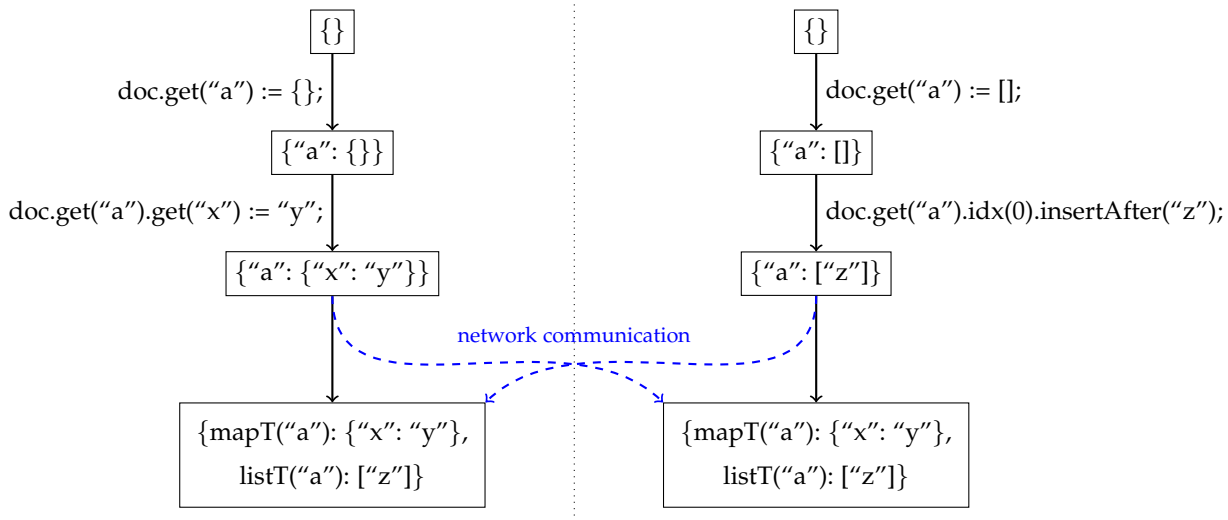

Fig. 5. Concurrently assigning values of different types to the same map key.

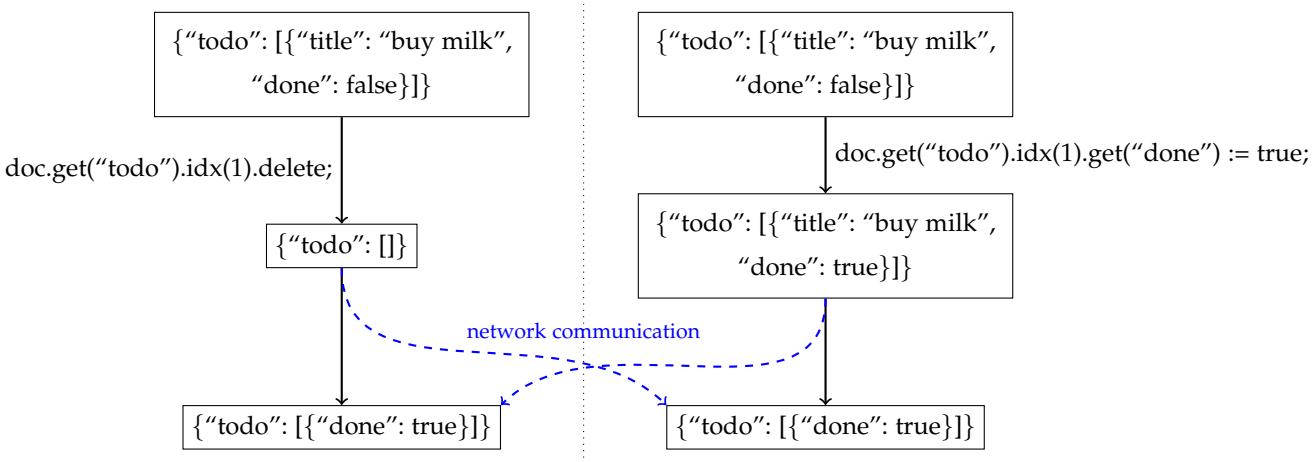

Fig. 6. One replica removes a list element, while another concurrently updates its contents.

would be overridden by the blanking-out of the entire map of colors. However, that behavior would violate our requirement that no user input should be lost due to concurrent modifications. Instead, we define merge semantics that preserve all changes, as shown in Figure 2: "blue" must be absent from the final map, since it was removed by blanking out the map, while "red" and "green" must be present, since they were explicitly added. This behavior matches that of CRDT maps in Riak [24], [25].

Figure 3 illustrates another problem with maps: two replicas can concurrently insert the same map key. Here, p and q each independently create a new shopping list under the same map key "grocery", and add items to the list. In the case of Figure 1, concurrent assignments to the same map key were left to be resolved by the application, but in Figure 3, both values are lists and so they can be merged automatically. We preserve the ordering and adjacency of items inserted at each replica, so "ham" appears after "eggs", and "flour" appears after "milk" in the merged result. There is no information on which replica's items should appear first in the merged result, so the algorithm can make an arbitrary choice between "eggs, ham, milk, flour" and "milk, flour, eggs, ham", provided that all replicas end up with the items in the same order.

Figure 4 shows how a collaborative text editor can be implemented, by treating the document as a list of characters. All changes are preserved in the merged result: "y" is inserted before "a"; "x" and "z" are inserted between "a" and "c"; and "b" is deleted.

Figure 5 demonstrates a variant of the situation in Figure 3, where two replicas concurrently insert the same map key, but they do so with different datatypes as values: p inserts a nested map, whereas q inserts a list. These datatypes cannot be meaningfully merged, so we preserve both values separately. We do this by tagging each map key with a type annotation (mapT, listT, or regT for a map, list, or register value respectively), so each type inhabits a separate namespace.

Finally, Figure 6 shows a limitation of the principle of preserving all user input. In a to-do list application, one replica removes a to-do item from the list, while another replica concurrently marks the same item as done. As the changes are merged, the update of the map key "done" effectively causes the list item to be resurrected on replica p, leaving a to-do item without a title (since the title was deleted as part of deleting the list item). This behavior is consistent with the example in Figure 2, but it is perhaps surprising. In this case it may be more desirable to discard one of the concurrent updates, and thus preserve the implicit schema that a to-do item has both a "title" and a "done" field. We leave the analysis of developer expectations and the development of a schema language for future work.

3.2 JSON Versus XML

The most common alternative to JSON is XML, and collaborative editing of XML documents has been previously studied [9], [10], [11]. Besides the superficial syntactical differences, the tree structure of XML and JSON appears quite similar. However, there is an important difference that we should highlight.

CMD	::=	`let` x = EXPR	$x \in$ VAR
	|	EXPR `:=` v	$v \in$ VAL
	|	EXPR`.insertAfter(`v`)`	$v \in$ VAL
	|	EXPR`.delete`	
	|	`yield`	
	|	CMD`;` CMD	
EXPR	::=	`doc`	
	|	x	$x \in$ VAR
	|	EXPR`.get(`key`)`	$key \in$ String
	|	EXPR`.idx(`i`)`	$i \geq 0$
	|	EXPR`.keys`	
	|	EXPR`.values`	
VAR	::=	x	$x \in$ VarString
VAL	::=	n	$n \in$ Number
	|	str	$str \in$ String
	|	`true` | `false` | `null`	
	|	`{}` | `[]`	

Fig. 7. Syntax of command language for querying and modifying a document.

```
doc := {};
doc.get("shopping") := [];
let head = doc.get("shopping").idx(0);
head.insertAfter("eggs");
let eggs = doc.get("shopping").idx(1);
head.insertAfter("cheese");
eggs.insertAfter("milk");

// Final state:
{"shopping": ["cheese", "eggs", "milk"]}
```

Fig. 8. Example of programmatically constructing a JSON document.

JSON has two collection constructs that can be arbitrarily nested: maps for unordered key-value pairs, and lists for ordered sequences. In XML, the children of an element form an ordered sequence, while the attributes of an element are unordered key-value pairs. However, XML does not allow nested elements inside attributes – the value of an attribute can only be a primitive datatype. Thus, XML supports maps within lists, but not lists within maps. In this regard, XML is less expressive than JSON: the scenarios in Figures 3 and 5 cannot occur in XML.

Some applications may attach map-like semantics to the children of an XML document, for example by interpreting the child element name as key. However, this key-value structure is not part of XML itself, and would not be enforced by existing collaborative editing algorithms for XML. If multiple children with the same key are concurrently created, existing algorithms would create duplicate children with the same key rather than merging them like in Figure 3.

3.3 Document Editing API

To define the semantics for collaboratively editable data structures, we first define a simple command language that is executed locally at any of the replicas, and which allows that replica's local copy of the document to be queried and modified. Performing read-only queries has no side-effects,

but modifying the document has the effect of producing *operations* describing the mutation. Those operations are immediately applied to the local copy of the document, and also enqueued for asynchronous broadcasting to other replicas.

The syntax of the command language is given in Figure 7. It is not a full programming language, but rather an API through which the document state is queried and modified. We assume that the program accepts user input and issues a (possibly infinite) sequence of commands to the API. We model only the semantics of those commands, and do not assume anything about the program in which the command language is embedded. The API differs slightly from the JSON libraries found in many programming languages, in order to allow us to define consistent merge semantics.

We first explain the language informally, before giving its formal semantics. The expression construct EXPR is used to construct a *cursor* which identifies a position in the document. An expression starts with either the special token `doc`, identifying the root of the JSON document tree, or a variable x that was previously defined in a `let` command. The expression defines, left to right, the path the cursor takes as it navigates through the tree towards the leaves: the operator `.get(`key`)` selects a key within a map, and `.idx(`n`)` selects the nth element of an ordered list. Lists are indexed starting from 1, and `.idx(0)` is a special cursor indicating the head of a list (a virtual position before the first list element).

The expression construct EXPR can also query the state of the document: `keys` returns the set of keys in the map at the current cursor, and `values` returns the contents of the multi-value register at the current cursor. (`values` is not defined if the cursor refers to a map or list.)

A command CMD either sets the value of a local variable (`let`), performs network communication (`yield`), or modifies the document. A document can be modified by writing to a register (the operator `:=` assigns the value of the register identified by the cursor), by inserting an element into a list (`insertAfter` places a new element after the existing list element identified by the cursor, and `.idx(0).insertAfter` inserts at the head of a list), or by deleting an element from a list or a map (`delete` removes the element identified by the cursor).

Figure 8 shows an example sequence of commands that constructs a new document representing a shopping list. First `doc` is set to `{}`, the empty map literal, and then the key `"shopping"` within that map is set to the empty list `[]`. The third line navigates to the key `"shopping"` and selects the head of the list, placing the cursor in a variable called `head`. The list element "eggs" is inserted at the head of the list. In line 5, the variable `eggs` is set to a cursor pointing at the list element "eggs". Then two more list elements are inserted: "cheese" at the head, and "milk" after "eggs".

Note that the cursor `eggs` identifies the list element by identity, not by its index: after the insertion of "cheese", the element "eggs" moves from index 1 to 2, but "milk" is nevertheless inserted after "eggs". As we shall see later, this feature is helpful for achieving desirable semantics in the presence of concurrent modifications.

4 Formal Semantics

We now explain formally how to achieve the concurrent semantics outlined in Section 3. The state of replica p is described by A_p, a finite partial function. The evaluation rules of the command language inspect and modify this local state A_p, and they do not depend on A_q (the state of any other replica $q \neq p$). The only communication between replicas occurs in the evaluation of the yield command, which we discuss later. For now, we concentrate on the execution of commands at a single replica p.

4.1 Expression Evaluation

Figure 9 gives the rules for evaluating EXPR expressions in the command language, which are evaluated in the context of the local replica state A_p. The EXEC rule formalizes the assumption that commands are executed sequentially. The LET rule allows the program to define a local variable, which is added to the local state (the notation $A_p[x \mapsto cur]$ denotes a partial function that is the same as A_p, except that $A_p(x) = cur$). The corresponding VAR rule allows the program to retrieve the value of a previously defined variable.

The following rules in Figure 9 show how an expression is evaluated to a cursor, which unambiguously identifies a particular position in a JSON document by describing a path from the root of the document tree to some branch or leaf node. A cursor consists only of immutable keys and identifiers, so it can be sent over the network to another replica, where it can be used to locate the same position in the document.

For example,

$$\mathsf{cursor}(\langle \mathsf{mapT}(\mathsf{doc}), \mathsf{listT}(\text{“shopping”})\rangle,\ id_1)$$

is a cursor representing the list element `"eggs"` in Figure 8, assuming that id_1 is the unique identifier of the operation that inserted this list element (we will discuss these identifiers in Section 4.2.1). The cursor can be interpreted as a path through the local replica state structure A_p, read from left to right: starting from the doc map at the root, it traverses through the map entry "shopping" of type listT, and finishes with the list element that has identifier id_1.

In general, $\mathsf{cursor}(\langle k_1, \ldots, k_{n-1}\rangle, k_n)$ consists of a (possibly empty) vector of keys $\langle k_1, \ldots, k_{n-1}\rangle$, and a final key k_n which is always present. k_n can be thought of as the final element of the vector, with the distinction that it is not tagged with a datatype, whereas the elements of the vector are tagged with the datatype of the branch node being traversed, either mapT or listT.

The DOC rule in Figure 9 defines the simplest cursor $\mathsf{cursor}(\langle\rangle, \mathsf{doc})$, referencing the root of the document using the special atom doc. The GET rule navigates a cursor to a particular key within a map. For example, the expression `doc.get("shopping")` evaluates to $\mathsf{cursor}(\langle \mathsf{mapT}(\mathsf{doc})\rangle, \text{“shopping”})$ by applying the DOC and GET rules. Note that the expression `doc.get(...)` implicitly asserts that doc is of type mapT, and this assertion is encoded in the cursor.

The rules $\textsc{Idx}_{1\ldots5}$ define how to evaluate the expression `.idx(`n`)`, moving the cursor to a particular element of a list. $\textsc{Idx}_1$ constructs a cursor representing the head of the

$$\textsc{Exec}\ \frac{cmd_1 : \mathrm{CMD} \qquad A_p,\ cmd_1 \implies A'_p}{A_p,\ \langle cmd_1\,;\, cmd_2\,;\,\ldots\rangle \implies A'_p,\ \langle cmd_2\,;\,\ldots\rangle} \qquad \textsc{Doc}\ \frac{}{A_p,\ \mathsf{doc} \implies \mathsf{cursor}(\langle\rangle,\ \mathsf{doc})}$$

$$\textsc{Let}\ \frac{A_p,\ expr \implies cur}{A_p,\ \mathsf{let}\ x = expr \implies A_p[\,x \mapsto cur\,]} \qquad \textsc{Var}\ \frac{x \in \mathrm{dom}(A_p)}{A_p,\ x \implies A_p(x)}$$

$$\textsc{Get}\ \frac{A_p,\ expr \implies \mathsf{cursor}(\langle k_1,\ldots,k_{n-1}\rangle,\ k_n) \qquad k_n \neq \mathsf{head}}{A_p,\ expr.\mathsf{get}(key) \implies \mathsf{cursor}(\langle k_1,\ldots,k_{n-1},\mathsf{mapT}(k_n)\rangle,\ key)}$$

$$\textsc{Idx}_1\ \frac{A_p,\ expr \implies \mathsf{cursor}(\langle k_1,\ldots,k_{n-1}\rangle,\ k_n) \qquad A_p,\ \mathsf{cursor}(\langle k_1,\ldots,k_{n-1},\mathsf{listT}(k_n)\rangle,\ \mathsf{head}).\mathsf{idx}(i) \implies cur'}{A_p,\ expr.\mathsf{idx}(i) \implies cur'}$$

$$\textsc{Idx}_2\ \frac{k_1 \in \mathrm{dom}(ctx) \qquad ctx(k_1),\ \mathsf{cursor}(\langle k_2,\ldots,k_{n-1}\rangle,\ k_n).\mathsf{idx}(i) \implies \mathsf{cursor}(\langle k_2,\ldots,k_{n-1}\rangle,\ k'_n)}{ctx,\ \mathsf{cursor}(\langle k_1,k_2,\ldots,k_{n-1}\rangle,\ k_n).\mathsf{idx}(i) \implies \mathsf{cursor}(\langle k_1,k_2,\ldots,k_{n-1}\rangle,\ k'_n)}$$

$$\textsc{Idx}_3\ \frac{i > 0 \ \wedge\ ctx(\mathsf{next}(k)) = k' \ \wedge\ k' \neq \mathsf{tail} \qquad ctx(\mathsf{pres}(k')) \neq \{\} \qquad ctx,\ \mathsf{cursor}(\langle\rangle,\ k').\mathsf{idx}(i-1) \implies ctx'}{ctx,\ \mathsf{cursor}(\langle\rangle,\ k).\mathsf{idx}(i) \implies ctx'}$$

$$\textsc{Idx}_4\ \frac{i > 0 \ \wedge\ ctx(\mathsf{next}(k)) = k' \ \wedge\ k' \neq \mathsf{tail} \qquad ctx(\mathsf{pres}(k')) = \{\} \qquad ctx,\ \mathsf{cursor}(\langle\rangle,\ k').\mathsf{idx}(i) \implies cur'}{ctx,\ \mathsf{cursor}(\langle\rangle,\ k).\mathsf{idx}(i) \implies cur'}$$

$$\textsc{Idx}_5\ \frac{i = 0}{ctx,\ \mathsf{cursor}(\langle\rangle,\ k).\mathsf{idx}(i) \implies \mathsf{cursor}(\langle\rangle,\ k)}$$

$$\mathrm{keys}(ctx) = \{\ k \mid \mathsf{mapT}(k) \in \mathrm{dom}(ctx) \ \vee\ \mathsf{listT}(k) \in \mathrm{dom}(ctx) \ \vee\ \mathsf{regT}(k) \in \mathrm{dom}(ctx)\ \}$$

$$\textsc{Keys}_1\ \frac{A_p,\ expr \implies cur \qquad A_p,\ cur.\mathsf{keys} \implies keys}{A_p,\ expr.\mathsf{keys} \implies keys}$$

$$\textsc{Keys}_2\ \frac{map = ctx(\mathsf{mapT}(k)) \qquad keys = \{\ k \mid k \in \mathrm{keys}(map) \ \wedge\ map(\mathsf{pres}(k)) \neq \{\}\ \}}{A_p,\ \mathsf{cursor}(\langle\rangle,\ k).\mathsf{keys} \implies keys}$$

$$\textsc{Keys}_3\ \frac{k_1 \in \mathrm{dom}(ctx) \qquad ctx(k_1),\ \mathsf{cursor}(\langle k_2,\ldots,k_{n-1}\rangle,\ k_n).\mathsf{keys} \implies keys}{ctx,\ \mathsf{cursor}(\langle k_1,k_2,\ldots,k_{n-1}\rangle,\ k_n).\mathsf{keys} \implies keys}$$

$$\textsc{Val}_1\ \frac{A_p,\ expr \implies cur \qquad A_p,\ cur.\mathsf{values} \implies val}{A_p,\ expr.\mathsf{values} \implies val}$$

$$\textsc{Val}_2\ \frac{\mathsf{regT}(k) \in \mathrm{dom}(ctx) \qquad val = \mathrm{range}(ctx(\mathsf{regT}(k)))}{ctx,\ \mathsf{cursor}(\langle\rangle,\ k).\mathsf{values} \implies val}$$

$$\textsc{Val}_3\ \frac{k_1 \in \mathrm{dom}(ctx) \qquad ctx(k_1),\ \mathsf{cursor}(\langle k_2,\ldots,k_{n-1}\rangle,\ k_n).\mathsf{values} \implies val}{ctx,\ \mathsf{cursor}(\langle k_1,k_2,\ldots,k_{n-1}\rangle,\ k_n).\mathsf{values} \implies val}$$

Fig. 9. Rules for evaluating expressions.

list, and delegates to the subsequent rules to scan over the list. IDX$_2$ recursively descends the local state according to the vector of keys in the cursor. When the vector of keys is empty, the context ctx is the subtree of A_p that stores the list in question, and the rules IDX$_{3,4,5}$ iterate over that list until the desired element is found.

IDX$_5$ terminates the iteration when the index reaches zero, while IDX$_3$ moves to the next element and decrements the index, and IDX$_4$ skips over list elements that are marked as deleted. The structure resembles a linked list: each list element has a unique identifier k, and the partial function representing local state maps $\mathsf{next}(k)$ to the ID of the list element that follows k.

Deleted elements are never removed from the linked list structure, but only marked as deleted (they become so-called *tombstones*). To this end, the local state maintains a *presence set* $\mathsf{pres}(k)$ for the list element with ID k, which is the set of all operations that have asserted the existence of this list element. When a list element is deleted, the presence set is set to the empty set, which marks it as deleted; however, a concurrent operation that references the list element can cause the presence set to be come non-empty again (leading to the situations in Figures 2 and 6). Rule IDX$_4$ handles list elements with an empty presence set by moving to the next list element without decrementing the index (i.e., not counting them as list elements).

The KEYS$_{1,2,3}$ rules allow the application to inspect the set of keys in a map. This set is determined by examining the local state, and excluding any keys for which the presence set is empty (indicating that the key has been deleted).

Finally, the $\text{VAL}_{1,2,3}$ rules allow the application to read the contents of a register at a particular cursor position, using a similar recursive rule structure as the IDX rules. A register is expressed using the regT type annotation in the local state. Although a replica can only assign a single value to a register, a register can nevertheless contain multiple values if multiple replicas concurrently assign values to it.

4.2 Generating Operations

When commands mutate the state of the document, they generate *operations* that describe the mutation. In our semantics, a command never directly modifies the local replica state A_p, but only generates an operation. That operation is then immediately applied to A_p so that it takes effect locally, and the same operation is also asynchronously broadcast to the other replicas.

4.2.1 Lamport Timestamps

Every operation in our model is given a unique identifier, which is used in the local state and in cursors. Whenever an element is inserted into a list, or a value is assigned to a register, the new list element or register value is identified by the identifier of the operation that created it.

In order to generate globally unique operation identifiers without requiring synchronous coordination between replicas we use Lamport timestamps [42]. A Lamport timestamp is a pair (c, p) where $p \in \text{ReplicaID}$ is the unique identifier of the replica on which the edit is made (for example, a hash of its public key), and $c \in \mathbb{N}$ is a counter that is stored at each replica and incremented for every operation. Since each replica generates a strictly monotonically increasing sequence of counter values c, the pair (c, p) is unique.

If a replica receives an operation with a counter value c that is greater than the locally stored counter value, the local counter is increased to the value of the incoming counter. This ensures that if operation o_1 causally happened before o_2 (that is, the replica that generated o_2 had received and processed o_1 before o_2 was generated), then o_2 must have a greater counter value than o_1. Only concurrent operations can have equal counter values.

We can thus define a total ordering $<$ for Lamport timestamps:

$$(c_1, p_1) < (c_2, p_2) \text{ iff } (c_1 < c_2) \vee (c_1 = c_2 \wedge p_1 < p_2).$$

If one operation happened before another, this ordering is consistent with causality (the earlier operation has a lower timestamp). If two operations are concurrent, their order according to $<$ is arbitrary but deterministic. This ordering property is important for our definition of the semantics of ordered lists.

4.2.2 Operation Structure

An operation is a tuple of the form

$$\begin{aligned}
&\mathsf{op}(\\
&\quad id : \mathbb{N} \times \text{ReplicaID},\\
&\quad deps : \mathcal{P}(\mathbb{N} \times \text{ReplicaID}),\\
&\quad cur : \mathsf{cursor}(\langle k_1, \ldots, k_{n-1} \rangle, k_n),\\
&\quad mut : \mathsf{insert}(v) \mid \mathsf{delete} \mid \mathsf{assign}(v) \quad v : \text{VAL}\\
&)
\end{aligned}$$

where id is the Lamport timestamp that uniquely identifies the operation, cur is the cursor describing the position in the document being modified, and mut is the mutation that was requested at the specified position.

$deps$ is the set of *causal dependencies* of the operation. It is defined as follows: if operation o_2 was generated by replica p, then a causal dependency of o_2 is any operation o_1 that had already been applied on p at the time when o_2 was generated. In this paper, we define $deps$ as the set of Lamport timestamps of all causal dependencies. In a real implementation, this set would become impracticably large, so a compact representation of causal history would be used instead – for example, version vectors [43], state vectors [5], or dotted version vectors [44]. However, to avoid ambiguity in our semantics we give the dependencies as a simple set of operation IDs.

The purpose of the causal dependencies $deps$ is to impose a partial ordering on operations: an operation can only be applied after all operations that "happened before" it have been applied. In particular, this means that the sequence of operations generated at one particular replica will be applied in the same order at every other replica. Operations that are concurrent (i.e., where there is no causal dependency in either direction) can be applied in any order.

4.2.3 Semantics of Generating Operations

The evaluation rules for commands are given in Figure 10. The MAKE-ASSIGN, MAKE-INSERT and MAKE-DELETE rules define how these respective commands mutate the document: all three delegate to the MAKE-OP rule to generate and apply the operation. MAKE-OP generates a new Lamport timestamp by choosing a counter value that is 1 greater than any existing counter in $A_p(\mathsf{ops})$, the set of all operation IDs that have been applied to replica p.

MAKE-OP constructs an op() tuple of the form described above, and delegates to the APPLY-LOCAL rule to process the operation. APPLY-LOCAL does three things: it evaluates the operation to produce a modified local state A'_p, it adds the operation to the queue of generated operations $A_p(\mathsf{queue})$, and it adds the ID to the set of processed operations $A_p(\mathsf{ops})$.

The yield command, inspired by Burckhardt et al. [35], performs network communication: sending and receiving operations to and from other replicas, and applying operations from remote replicas. The rules APPLY-REMOTE, SEND, RECV and YIELD define the semantics of yield. Since any of these rules can be used to evaluate yield, their effect is nondeterministic, which models the asynchronicity of the network: a message sent by one replica arrives at another replica at some arbitrarily later point in time, and there is no message ordering guarantee in the network.

The SEND rule takes any operations that were placed in $A_p(\mathsf{queue})$ by APPLY-LOCAL and adds them to a send buffer $A_p(\mathsf{send})$. Correspondingly, the RECV rule takes operations in the send buffer of replica q and adds them to the receive buffer $A_p(\mathsf{recv})$ of replica p. This is the only rule that involves more than one replica, and it models all network communication.

Once an operation appears in the receive buffer $A_p(\mathsf{recv})$, the rule APPLY-REMOTE may apply. Under the preconditions that the operation has not already been processed and

$$\textsc{Make-Assign}\ \frac{A_p,\ expr \implies cur \qquad val : \mathrm{VAL} \qquad A_p,\ \mathsf{makeOp}(cur, \mathsf{assign}(val)) \implies A'_p}{A_p,\ expr := val \implies A'_p}$$

$$\textsc{Make-Insert}\ \frac{A_p,\ expr \implies cur \qquad val : \mathrm{VAL} \qquad A_p,\ \mathsf{makeOp}(cur, \mathsf{insert}(val)) \implies A'_p}{A_p,\ expr.\mathsf{insertAfter}(val) \implies A'_p}$$

$$\textsc{Make-Delete}\ \frac{A_p,\ expr \implies cur \qquad A_p,\ \mathsf{makeOp}(cur, \mathsf{delete}) \implies A'_p}{A_p,\ expr.\mathsf{delete} \implies A'_p}$$

$$\textsc{Make-Op}\ \frac{ctr = \max(\{0\} \cup \{c_i \mid (c_i, p_i) \in A_p(\mathsf{ops})\} \qquad A_p,\ \mathsf{apply}(\mathsf{op}((ctr+1, p), A_p(\mathsf{ops}), cur, mut)) \implies A'_p}{A_p,\ \mathsf{makeOp}(cur, mut) \implies A'_p}$$

$$\textsc{Apply-Local}\ \frac{A_p,\ op \implies A'_p}{A_p,\ \mathsf{apply}(op) \implies A'_p[\,\mathsf{queue} \mapsto A'_p(\mathsf{queue}) \cup \{op\},\ \mathsf{ops} \mapsto A'_p(\mathsf{ops}) \cup \{op.id\}\,]}$$

$$\textsc{Apply-Remote}\ \frac{op \in A_p(\mathsf{recv}) \qquad op.id \notin A_p(\mathsf{ops}) \qquad op.deps \subseteq A_p(\mathsf{ops}) \qquad A_p,\ op \implies A'_p}{A_p,\ \mathsf{yield} \implies A'_p[\,\mathsf{ops} \mapsto A'_p(\mathsf{ops}) \cup \{op.id\}\,]}$$

$$\textsc{Send}\ \frac{}{A_p,\ \mathsf{yield} \implies A_p[\,\mathsf{send} \mapsto A_p(\mathsf{send}) \cup A_p(\mathsf{queue})\,]}$$

$$\textsc{Recv}\ \frac{q : \mathrm{ReplicaID}}{A_p,\ \mathsf{yield} \implies A_p[\,\mathsf{recv} \mapsto A_p(\mathsf{recv}) \cup A_q(\mathsf{send})\,]}$$

$$\textsc{Yield}\ \frac{A_p,\ \mathsf{yield} \implies A'_p \qquad A'_p,\ \mathsf{yield} \implies A''_p}{A_p,\ \mathsf{yield} \implies A''_p}$$

Fig. 10. Rules for generating, sending, and receiving operations.

that its causal dependencies are satisfied, APPLY-REMOTE applies the operation in the same way as APPLY-LOCAL, and adds its ID to the set of processed operations $A_p(\mathsf{ops})$.

The actual document modifications are performed by applying the operations, which we discuss next.

4.3 Applying Operations

Figure 11 gives the rules that apply an operation op to a context ctx, producing an updated context ctx'. The context is initially the replica state A_p, but may refer to subtrees of the state as rules are recursively applied. These rules are used by APPLY-LOCAL and APPLY-REMOTE to perform the state updates on a document.

When the operation cursor's vector of keys is non-empty, the DESCEND rule first applies. It recursively descends the document tree by following the path described by the keys. If the tree node already exists in the local replica state, CHILD-GET finds it, otherwise CHILD-MAP and CHILD-LIST create an empty map or list respectively.

The DESCEND rule also invokes ADD-ID$_{1,2}$ at each tree node along the path described by the cursor, adding the operation ID to the presence set $\mathsf{pres}(k)$ to indicate that the subtree includes a mutation made by this operation.

The remaining rules in Figure 11 apply when the vector of keys in the cursor is empty, which is the case when descended to the context of the tree node to which the mutation applies. The ASSIGN rule handles assignment of a primitive value to a register, EMPTY-MAP handles assignment where the value is the empty map literal `{}`, and EMPTY-LIST handles assignment of the empty list `[]`. These three rules for assign have a similar structure: first clearing the prior value at the cursor (as discussed in the next section), then adding the operation ID to the presence set, and finally incorporating the new value into the tree of local state.

The INSERT$_{1,2}$ rules handle insertion of a new element into an ordered list. In this case, the cursor refers to the list element $prev$, and the new element is inserted after that position in the list. INSERT$_1$ performs the insertion by manipulating the linked list structure. INSERT$_2$ handles the case of multiple replicas concurrently inserting list elements at the same position, and uses the ordering relation $<$ on Lamport timestamps to consistently determine the insertion point. Our approach for handling insertions is based on the RGA algorithm [27]. We show later that these rules ensure all replicas converge towards the same state.

4.3.1 Clearing Prior State

Assignment and deletion operations require that prior state (the value being overwritten or deleted) is cleared, while also ensuring that concurrent modifications are not lost, as illustrated in Figure 2. The rules to handle this clearing process are given in Figure 12. Intuitively, the effect of clearing something is to reset it to its empty state by undoing any operations that causally precede the current operation, while leaving the effect of any concurrent operations untouched.

A delete operation can be used to delete either an element from an ordered list or a key from a map, depending on what the cursor refers to. The DELETE rule shows how

$$\textsc{Descend}\ \frac{ctx,\ k_1 \implies child \qquad child,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle k_2, \ldots, k_{n-1}\rangle,\ k_n), mut) \implies child' \qquad ctx,\ \mathsf{addId}(k_1, id, mut) \implies ctx'}{ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle k_1, k_2, \ldots, k_{n-1}\rangle,\ k_n), mut) \implies ctx'[\,k_1 \mapsto child'\,]}$$

$$\textsc{Child-Get}\ \frac{k \in \mathrm{dom}(ctx)}{ctx,\ k \implies ctx(k)} \qquad \textsc{Child-Map}\ \frac{\mathsf{mapT}(k) \notin \mathrm{dom}(ctx)}{ctx,\ \mathsf{mapT}(k) \implies \{\}} \qquad \textsc{Child-List}\ \frac{\mathsf{listT}(k) \notin \mathrm{dom}(ctx)}{ctx,\ \mathsf{listT}(k) \implies \{\,\mathsf{next}(\mathsf{head}) \mapsto \mathsf{tail}\,\}}$$

$$\textsc{Child-Reg}\ \frac{\mathsf{regT}(k) \notin \mathrm{dom}(ctx)}{ctx,\ \mathsf{regT}(k) \implies \{\}} \qquad \textsc{Presence}_1\ \frac{\mathsf{pres}(k) \in \mathrm{dom}(ctx)}{ctx,\ \mathsf{pres}(k) \implies ctx(\mathsf{pres}(k))} \qquad \textsc{Presence}_2\ \frac{\mathsf{pres}(k) \notin \mathrm{dom}(ctx)}{ctx,\ \mathsf{pres}(k) \implies \{\}}$$

$$\textsc{Add-ID}_1\ \frac{mut \neq \mathsf{delete} \qquad k_{tag} \in \{\mathsf{mapT}(k), \mathsf{listT}(k), \mathsf{regT}(k)\} \qquad ctx,\ \mathsf{pres}(k) \implies pres}{ctx,\ \mathsf{addId}(k_{tag}, id, mut) \implies ctx[\,\mathsf{pres}(k) \mapsto pres \cup \{id\}\,]} \qquad \textsc{Add-ID}_2\ \frac{mut = \mathsf{delete}}{ctx,\ \mathsf{addId}(k_{tag}, id, mut) \implies ctx}$$

$$\textsc{Assign}\ \frac{val \neq [\,] \wedge val \neq \{\,\} \qquad ctx,\ \mathsf{clear}(deps, \mathsf{regT}(k)) \implies ctx',\ pres \qquad ctx',\ \mathsf{addId}(\mathsf{regT}(k), id, \mathsf{assign}(val)) \implies ctx'' \qquad ctx'',\ \mathsf{regT}(k) \implies child}{ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle,\ k), \mathsf{assign}(val)) \implies ctx''[\,\mathsf{regT}(k) \mapsto child[\,id \mapsto val\,]\,]}$$

$$\textsc{Empty-Map}\ \frac{val = \{\,\} \qquad ctx,\ \mathsf{clearElem}(deps, k) \implies ctx',\ pres \qquad ctx',\ \mathsf{addId}(\mathsf{mapT}(k), id, \mathsf{assign}(val)) \implies ctx'' \qquad ctx'',\ \mathsf{mapT}(k) \implies child}{ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle,\ k), \mathsf{assign}(val)) \implies ctx''[\,\mathsf{mapT}(k) \mapsto child\,]}$$

$$\textsc{Empty-List}\ \frac{val = [\,] \qquad ctx,\ \mathsf{clearElem}(deps, k) \implies ctx',\ pres \qquad ctx',\ \mathsf{addId}(\mathsf{listT}(k), id, \mathsf{assign}(val)) \implies ctx'' \qquad ctx'',\ \mathsf{listT}(k) \implies child}{ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle,\ k), \mathsf{assign}(val)) \implies ctx''[\,\mathsf{listT}(k) \mapsto child\,]}$$

$$\textsc{Insert}_1\ \frac{ctx(\mathsf{next}(prev)) = next \qquad next < id \vee next = \mathsf{tail} \qquad ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle,\ id), \mathsf{assign}(val)) \implies ctx'}{ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle,\ prev), \mathsf{insert}(val)) \implies ctx'[\,\mathsf{next}(prev) \mapsto id,\ \mathsf{next}(id) \mapsto next\,]}$$

$$\textsc{Insert}_2\ \frac{ctx(\mathsf{next}(prev)) = next \qquad id < next \qquad ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle,\ next), \mathsf{insert}(val)) \implies ctx'}{ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle,\ prev), \mathsf{insert}(val)) \implies ctx'}$$

Fig. 11. Rules for applying insertion and assignment operations to update the state of a replica.

$$\textsc{Delete}\ \frac{ctx,\ \mathsf{clearElem}(deps, k) \implies ctx',\ pres}{ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle,\ k), \mathsf{delete}) \implies ctx'}$$

$$\textsc{Clear-Elem}\ \frac{ctx,\ \mathsf{clearAny}(deps, k) \implies ctx', pres_1 \qquad ctx',\ \mathsf{pres}(k) \implies pres_2 \qquad pres_3 = pres_1 \cup pres_2 \setminus deps}{ctx,\ \mathsf{clearElem}(deps, k) \implies ctx'[\,\mathsf{pres}(k) \mapsto pres_3\,],\ pres_3}$$

$$\textsc{Clear-Any}\ \frac{\begin{array}{c}ctx,\ \mathsf{clear}(deps, \mathsf{mapT}(k)) \\ \implies ctx_1,\ pres_1\end{array} \qquad \begin{array}{c}ctx_1,\ \mathsf{clear}(deps, \mathsf{listT}(k)) \\ \implies ctx_2,\ pres_2\end{array} \qquad \begin{array}{c}ctx_2,\ \mathsf{clear}(deps, \mathsf{regT}(k)) \\ \implies ctx_3,\ pres_3\end{array}}{ctx,\ \mathsf{clearAny}(deps, k) \implies ctx_3,\ pres_1 \cup pres_2 \cup pres_3}$$

$$\textsc{Clear-None}\ \frac{k \notin \mathrm{dom}(ctx)}{ctx,\ \mathsf{clear}(deps, k) \implies ctx,\ \{\}}$$

$$\textsc{Clear-Reg}\ \frac{\mathsf{regT}(k) \in \mathrm{dom}(ctx) \qquad concurrent = \{id \mapsto v \mid (id \mapsto v) \in ctx(\mathsf{regT}(k)) \wedge id \notin deps\}}{ctx,\ \mathsf{clear}(deps, \mathsf{regT}(k)) \implies ctx[\,\mathsf{regT}(k) \mapsto concurrent\,],\ \mathrm{dom}(concurrent)}$$

$$\textsc{Clear-Map}_1\ \frac{\mathsf{mapT}(k) \in \mathrm{dom}(ctx) \qquad ctx(\mathsf{mapT}(k)),\ \mathsf{clearMap}(deps, \{\}) \implies cleared,\ pres}{ctx,\ \mathsf{clear}(deps, \mathsf{mapT}(k)) \implies ctx[\,\mathsf{mapT}(k) \mapsto cleared\,],\ pres}$$

$$\textsc{Clear-Map}_2\ \frac{\begin{array}{c}k \in \mathrm{keys}(ctx) \\ \wedge\, k \notin done\end{array} \qquad \begin{array}{c}ctx,\ \mathsf{clearElem}(deps, k) \\ \implies ctx',\ pres_1\end{array} \qquad \begin{array}{c}ctx',\ \mathsf{clearMap}(deps, done \cup \{k\}) \\ \implies ctx'',\ pres_2\end{array}}{ctx,\ \mathsf{clearMap}(deps, done) \implies ctx'',\ pres_1 \cup pres_2}$$

$$\textsc{Clear-Map}_3\ \frac{done = \mathrm{keys}(ctx)}{ctx,\ \mathsf{clearMap}(deps, done) \implies ctx,\ \{\}}$$

$$\textsc{Clear-List}_1\ \frac{\mathsf{listT}(k) \in \mathrm{dom}(ctx) \qquad ctx(\mathsf{listT}(k)),\ \mathsf{clearList}(deps, \mathsf{head}) \implies cleared,\ pres}{ctx,\ \mathsf{clear}(deps, \mathsf{listT}(k)) \implies ctx[\,\mathsf{listT}(k) \mapsto cleared\,],\ pres}$$

$$\textsc{Clear-List}_2\ \frac{\begin{array}{c}k \neq \mathsf{tail} \wedge \\ ctx(\mathsf{next}(k)) = next\end{array} \qquad \begin{array}{c}ctx,\ \mathsf{clearElem}(deps, k) \\ \implies ctx',\ pres_1\end{array} \qquad \begin{array}{c}ctx',\ \mathsf{clearList}(deps, next) \\ \implies ctx'',\ pres_2\end{array}}{ctx,\ \mathsf{clearList}(deps, k) \implies ctx'',\ pres_1 \cup pres_2}$$

$$\textsc{Clear-List}_3\ \frac{k = \mathsf{tail}}{ctx,\ \mathsf{clearList}(deps, k) \implies ctx,\ \{\}}$$

Fig. 12. Rules for applying deletion operations to update the state of a replica.

this operation is evaluated by delegating to CLEAR-ELEM. In turn, CLEAR-ELEM uses CLEAR-ANY to clear out any data with a given key, regardless of whether it is of type mapT, listT or regT, and also updates the presence set to include any nested operation IDs, but exclude any operations in $deps$.

The premises of CLEAR-ANY are satisfied by CLEAR-MAP$_1$, CLEAR-LIST$_1$ and CLEAR-REG if the respective key appears in ctx, or by CLEAR-NONE (which does nothing) if the key is absent.

As defined by the ASSIGN rule, a register maintains a mapping from operation IDs to values. CLEAR-REG updates a register by removing all operation IDs that appear in $deps$ (i.e., which causally precede the clearing operation), but retaining all operation IDs that do not appear in $deps$ (from assignment operations that are concurrent with the clearing operation).

Clearing maps and lists takes a similar approach: each element of the map or list is recursively cleared using clearElem, and presence sets are updated to exclude $deps$. Thus, any list elements or map entries whose modifications causally precede the clearing operation will end up with empty presence sets, and thus be considered deleted. Any map or list elements containing operations that are concurrent with the clearing operation are preserved.

4.4 Convergence

As outlined in Section 1.2, we require that all replicas automatically converge towards the same state – a key requirement of a CRDT. We now formalize this notion, and show that the rules in Figures 9 to 12 satisfy this requirement.

Definition 1 (valid execution)**.** *A* valid execution *is a set of operations generated by a set of replicas $\{p_1, \ldots, p_k\}$, each reducing a sequence of commands $\langle cmd_1 ; \ldots ; cmd_n \rangle$ without getting stuck.*

A reduction gets stuck if there is no application of rules in which all premises are satisfied. For example, the IDX$_{3,4}$ rules in Figure 9 get stuck if $\mathsf{idx}(n)$ tries to iterate past the end of a list, which would happen if n is greater than the number of non-deleted elements in the list; in a real implementation this would be a runtime error. By constraining valid executions to those that do not get stuck, we ensure that operations only refer to list elements that actually exist.

Note that it is valid for an execution to never perform any network communication, either because it never invokes the yield command, or because the nondeterministic execution of yield never applies the RECV rule. We need only a replica's local state to determine whether reduction gets stuck.

Definition 2 (history)**.** *A* history *is a sequence of operations in the order it was applied at one particular replica p by application of the rules* APPLY-LOCAL *and* APPLY-REMOTE*.*

Since the evaluation rules sequentially apply one operation at a time at a given replica, the order is well-defined. Even if two replicas p and q applied the same set of operations, i.e. if $A_p(\mathsf{ops}) = A_q(\mathsf{ops})$, they may have applied any concurrent operations in a different order. Due to the premise $op.deps \subseteq A_p(\mathsf{ops})$ in APPLY-REMOTE, histories are consistent with causality: if an operation has causal dependencies, it appears at some point after those dependencies in the history.

Definition 3 (document state)**.** *The* document state *of a replica p is the subtree of A_p containing the document: that is, $A_p(\mathsf{mapT(doc)})$ or $A_p(\mathsf{listT(doc)})$ or $A_p(\mathsf{regT(doc)})$, whichever is defined.*

A_p contains variables defined with let, which are local to one replica, and not part of the replicated state. The definition of document state excludes these variables.

Theorem. *For any two replicas p and q that participated in a valid execution, if $A_p(\mathsf{ops}) = A_q(\mathsf{ops})$, then p and q have the same document state.*

This theorem is proved in the appendix. It formalizes the safety property of convergence: if two replicas have processed the same set of operations, possibly in a different order, then they are in the same state. In combination with a liveness property, namely that every replica eventually processes all operations, we obtain the desired notion of convergence: all replicas eventually end up in the same state.

The liveness property depends on assumptions of replicas invoking yield sufficiently often, and all nondeterministic rules for yield being chosen fairly. We will not formalize the liveness property in this paper, but assert that it can usually be provided in practice, as network interruptions are usually of finite duration.

5 Conclusions and Further Work

In this paper we demonstrated how to compose CRDTs for ordered lists, maps and registers into a compound CRDT with a JSON data model. It supports arbitrarily nested lists and maps, and it allows replicas to make arbitrary changes to the data without waiting for network communication. Replicas asynchronously send mutations to other replicas in the form of operations. Concurrent operations are commutative, which ensures that replicas converge towards the same state without requiring application-specific conflict resolution logic.

This work focused on the formal semantics of the JSON CRDT, represented as a mathematical model. We are also working on a practical implementation of the algorithm, and will report on its performance characteristics in follow-on work.

Our principle of not losing input due to concurrent modifications appears at first glance to be reasonable, but as illustrated in Figure 6, it leads to merged document states that may be surprising to application programmers who are more familiar with sequential programs. Further work will be needed to understand the expectations of application programmers, and to design data structures that are minimally surprising under concurrent modification. It may turn out that a schema language will be required to support more complex applications. A schema language could also support semantic annotations, such as indicating that a number should be treated as a counter rather than a register.

The CRDT defined in this paper supports insertion, deletion and assignment operations. In addition to these, it would be useful to support a *move* operation (to change the order of elements in an ordered list, or to move a subtree from one position in a document to another) and an *undo* operation. Moreover, garbage collection (tombstone removal) is required in order to prevent unbounded growth of the data structure. We plan to address these missing features in future work.

Acknowledgements

This research was supported by a grant from The Boeing Company. Thank you to Dominic Orchard, Diana Vasile, and the anonymous reviewers for comments that improved this paper.

References

[1] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in partitioned networks," *ACM Computing Surveys*, vol. 17, no. 3, pp. 341–370, Sep. 1985.

[2] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith, "SoK: Secure messaging," in *36th IEEE Symposium on Security and Privacy*, May 2015.

[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Oct. 2011, pp. 386–400.

[4] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte, "Brief announcement: Semantics of eventually consistent replicated sets," in *26th International Symposium on Distributed Computing (DISC)*, Oct. 2012.

[5] C. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *ACM International Conference on Management of Data (SIGMOD)*, May 1989, pp. 399–407.

[6] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuer, "An integrating, transformation-oriented approach to concurrency control and undo in group editors," in *ACM Conference on Computer Supported Cooperative Work (CSCW)*, Nov. 1996, pp. 288–297.

[7] C. Sun and C. Ellis, "Operational transformation in real-time group editors: Issues, algorithms, and achievements," in *ACM Conference on Computer Supported Cooperative Work (CSCW)*, Nov. 1998, pp. 59–68.

[8] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping, "High-latency, low-bandwidth windowing in the Jupiter collaboration system," in *8th Annual ACM Symposium on User Interface Software and Technology (UIST)*, Nov. 1995, pp. 111–120.

[9] A. H. Davis, C. Sun, and J. Lu, "Generalizing operational transformation to the Standard General Markup Language," in *ACM Conference on Computer Supported Cooperative Work (CSCW)*, Nov. 2002, pp. 58–67.

[10] C.-L. Ignat and M. C. Norrie, "Customizable collaborative editor relying on treeOPT algorithm," in *8th European Conference on Computer-Supported Cooperative Work (ECSCW)*, Sep. 2003, pp. 315–334.

[11] D. Wang, A. Mah, S. Lassen, and S. Thorogood. (2015, Aug.) Apache Wave (incubating) protocol documentation, release 0.4. Apache Software Foundation. [Online]. Available: https://people.apache.org/~al/wave_docs/ApacheWaveProtocol-0.4.pdf

[12] D. Li and R. Li, "A performance study of group editing algorithms," in *12th International Conference on Parallel and Distributed Systems (ICPADS)*, Jul. 2006, pp. 300–307.

[13] A.-N. Mehdi, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso, "Evaluating CRDTs for real-time document editing," in *11th ACM Symposium on Document Engineering (DocEng)*, Sep. 2011, pp. 103–112.

[14] J. Day-Richter. (2010, Sep.) What's different about the new Google Docs: Making collaboration fast. [Online]. Available: https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html

[15] AppJet, Inc. (2011, Mar.) Etherpad and EasySync technical manual. [Online]. Available: https://github.com/ether/etherpad-lite/blob/e2ce9dc/doc/easysync/easysync-full-description.pdf

[16] D. Spiewak. (2010, May) Understanding and applying operational transformation. [Online]. Available: http://www.codecommit.com/blog/java/understanding-and-applying-operational-transformation

[17] M. Lemonik, Personal communication, Mar. 2016.

[18] A. Imine, P. Molli, G. Oster, and M. Rusinowitch, "Proving correctness of transformation functions in real-time groupware," in *8th European Conference on Computer-Supported Cooperative Work (ECSCW)*, Sep. 2003, pp. 277–293.

[19] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[20] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

[21] H. Attiya, F. Ellen, and A. Morrison, "Limitations of highly-available eventually-consistent data stores," in *ACM Symposium on Principles of Distributed Computing (PODC)*, Jul. 2015.

[22] Google, Inc. (2015) Google Realtime API. [Online]. Available: https://developers.google.com/google-apps/realtime/overview

[23] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," INRIA, Tech. Rep. 7506, 2011.

[24] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott, "Riak DT map: a composable, convergent replicated dictionary," in *1st Workshop on Principles and Practice of Eventual Consistency (PaPEC)*, Apr. 2014.

[25] R. Brown. (2013, Oct.) A bluffers guide to CRDTs in Riak. [Online]. Available: https://gist.github.com/russelldb/f92f44bdfb619e089a4d

[26] G. Oster, P. Urso, P. Molli, and A. Imine, "Data consistency for P2P collaborative editing," in *ACM Conference on Computer Supported Cooperative Work (CSCW)*, Nov. 2006.

[27] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 354–368, 2011.

[28] N. Preguiça, J. Manuel Marquès, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *29th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2009.

[29] S. Weiss, P. Urso, and P. Molli, "Logoot-Undo: Distributed collaborative editing system on P2P networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 8, pp. 1162–1174, Jan. 2010.

[30] B. Nédelec, P. Molli, A. Mostefaoui, and E. Desmontils, "LSEQ: an adaptive structure for sequences in distributed collaborative editing," in *13th ACM Symposium on Document Engineering (DocEng)*, Sep. 2013, pp. 37–46.

[31] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski, "Specification and complexity of collaborative text editing," in *ACM Symposium on Principles of Distributed Computing (PODC)*, Jul. 2016, pp. 259–268.

[32] C. Baquero, P. S. Almeida, and C. Lerche, "The problem with embedded CRDT counters and a solution," in *2nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC)*, Apr. 2016.

[33] P. S. Almeida, A. Shoker, and C. Baquero, "Delta state replicated data types," arXiv:1603.01529 [cs.DC], Mar. 2016. [Online]. Available: http://arxiv.org/abs/1603.01529

[34] C. Baquero, P. S. Almeida, A. Cunha, and C. Ferreira, "Composition of state-based CRDTs," HASLab, Tech. Rep., May 2015. [Online]. Available: http://haslab.uminho.pt/cbm/files/crdtcompositionreport.pdf

[35] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood, "Cloud types for eventual consistency," in *26th European Conference on Object-Oriented Programming (ECOOP)*, Jun. 2012.

[36] S. Martin, P. Urso, and S. Weiss, "Scalable XML collaborative editing with undo," in *On the Move to Meaningful Internet Systems*, Oct. 2010, pp. 507–514.

[37] 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," in *21st ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007, pp. 205–220.

[38] S. Balasubramaniam and B. C. Pierce, "What is a file synchronizer?" in *4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, Oct. 1998, pp. 98–108.

[39] N. Ramsey and E. Csirmaz, "An algebraic approach to file synchronization," in *8th European Software Engineering Conference (ESEC/FSE-9)*, Sep. 2001.

[40] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," in *15th ACM Symposium on Operating Systems Principles (SOSP)*, Dec. 1995, pp. 172–182.

[41] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 185–196, Nov. 2014.

[42] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[43] D. S. Parker, Jr, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of mutual inconsistency in distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 3, pp. 240–247, May 1983.

[44] N. Preguiça, C. Baquero, P. S. Almeida, V. Fonte, and R. Gonçalves, "Brief announcement: Efficient causality tracking in distributed storage systems with dotted version vectors," in *31st ACM Symposium on Principles of Distributed Computing (PODC)*, Jul. 2012.

Martin Kleppmann is a Research Associate in the Computer Laboratory at the University of Cambridge. His current research project, TRVE Data, is working towards better security and privacy in cloud applications by applying end-to-end encryption to collaboratively editable application data. His book *Designing Data-Intensive Applications* was published by O'Reilly Media in 2017. Previously, he worked as a software engineer and entrepreneur at several internet companies, including Rapportive and LinkedIn.

Alastair R. Beresford is a Senior Lecturer in the Computer Laboratory at the University of Cambridge. His research work explores the security and privacy of large-scale distributed systems, with a particular focus on networked mobile devices such as smartphones, tablets and laptops. He looks at the security and privacy of the devices themselves, as well as the security and privacy problems induced by the interaction between mobile devices and cloud-based Internet services.

APPENDIX
PROOF OF CONVERGENCE

Theorem 1. *For any two replicas p and q that participated in a valid execution, if $A_p(\mathsf{ops}) = A_q(\mathsf{ops})$, then p and q have the same document state.*

Proof. Consider the histories H_p and H_q at p and q respectively (see Definition 2). The rules APPLY-LOCAL and APPLY-REMOTE maintain the invariant that an operation is added to $A_p(\mathsf{ops})$ or $A_q(\mathsf{ops})$ if and only if it was applied to the document state at p or q. Thus, $A_p(\mathsf{ops}) = A_q(\mathsf{ops})$ iff H_p and H_q contain the same set of operations (potentially ordered differently).

The history H_p at replica p is a sequence of n operations: $H_p = \langle o_1, \ldots, o_n \rangle$, and the document state at p is derived from H_p by starting in the empty state and applying the operations in order. Likewise, the document state at q is derived from H_q, which is a permutation of H_p. Both histories must be consistent with causality, i.e. for all i with $1 \leq i \leq n$, we require $o_i.deps \subseteq \{o_j.id \mid 1 \leq j < i\}$. The causality invariant is maintained by the APPLY-* rules.

We can prove the theorem by induction over the length of history n.

Base case: An empty history with $n = 0$ describes the empty document state. The empty document is always the same, and so any two replicas that have not executed any operations are by definition in the same state.

Induction step: Given histories H_p and H_q of length n, such that $H_p = \langle o_1, \ldots, o_n \rangle$ and H_q is a permutation of H_p, and such that applying H_p results in the same document state as applying H_q, we can construct new histories H'_p and H'_q of length $n+1$ by inserting a new operation o_{n+1} at any causally ready position in H_p or H_q respectively. We must then show that for all the histories H'_p and H'_q constructed this way, applying the sequence of operations in order results in the same document state.

In order to prove the induction step, we examine the insertion of o_{n+1} into H_p and H_q. Each history can be split into a prefix, which is the minimal subsequence $\langle o_1, \ldots, o_j \rangle$ such that $o_{n+1}.deps \subseteq \{o_1.id, \ldots, o_j.id\}$, and a suffix, which is the remaining subsequence $\langle o_{j+1}, \ldots, o_n \rangle$. The prefix contains all operations that causally precede o_{n+1}, and possibly some operations that are concurrent with o_{n+1}; the suffix contains only operations that are concurrent with o_{n+1}. The earliest position where o_{n+1} can be inserted into the history is between the prefix and the suffix; the latest position is at the end of the suffix; or it could be inserted at any point within the suffix.

We need to show that the effect on the document state is the same, regardless of the position at which o_{n+1} is inserted, and regardless of whether it is inserted into H_p or H_q. We do this in Lemma 8 by showing that o_{n+1} is commutative with respect to all operations in the suffix, i.e. with respect to any operations that are concurrent to o_{n+1}. □

Before we can prove the commutativity of operations, we must first define some more terms and prove some preliminary lemmas.

Definition 4 (appearing after)**.** *In the ordered list ctx, list element k_j* appears after *list element k_1 if there exists a (possibly empty) sequence of list elements $k_2, \ldots, k_{j-1}$ such that for all i with $1 \leq i < j$, $ctx(\mathsf{next}(k_i)) = k_{i+1}$. Moreover, we say k_j appears* immediately after *k_1 if that sequence is empty, i.e. if $ctx(\mathsf{next}(k_1)) = k_j$.*

The definition of *appearing after* corresponds to the order in which the IDX rules iterate over the list.

Lemma 2. *If k_2 appears after k_1 in an ordered list, and the list is mutated according to the evaluation rules, k_2 also appears after k_1 in all later document states.*

Proof. The only rule that modifies the next pointers in the context is INSERT_1, and it inserts a new list element between two existing list elements (possibly head and/or tail). This modification preserves the appears-after relationship between any two existing list elements. Since no other rule affects the list order, appears-after is always preserved. □

Note that deletion of an element from a list does not remove it from the sequence of next pointers, but only clears its presence set $\mathsf{pres}(k)$.

Lemma 3. *If one replica inserts a list element k_{new} between k_1 and k_2, i.e. if k_{new} appears after k_1 in the list and k_2 appears after k_{new} in the list on the source replica after applying* APPLY-LOCAL*, then k_{new} appears after k_1 and k_2 appears after k_{new} on every other replica where that operation is applied.*

Proof. The rules for generating list operations ensure that k_1 is either head or an operation identifier, and k_2 is either tail or an operation identifier.

When the insertion operation is generated using the MAKE-OP rule, its operation identifier is given a counter value ctr that is greater than the counter of any existing operation ID in $A_p(\mathsf{ops})$. If k_2 is an operation identifier, we must have $k_2 \in A_p(\mathsf{ops})$, since both APPLY-LOCAL and APPLY-REMOTE add operation IDs to $A_p(\mathsf{ops})$ when applying an insertion. Thus, either $k_2 < k_{new}$ under the ordering relation $<$ for Lamport timestamps, or $k_2 = \mathsf{tail}$.

When the insertion operation is applied on another replica using APPLY-REMOTE and $\text{INSERT}_{1,2}$, k_2 appears after k_1 on that replica (by Lemma 2 and causality). The cursor of the operation is $\mathsf{cursor}(\langle \ldots \rangle, k_1)$, so the rules start iterating the list at k_1, and therefore k_{new} is inserted at some position after k_1.

If other concurrent insertions occurred between k_1 and k_2, their operation ID may be greater than or less than k_{new}, and thus either INSERT_1 or INSERT_2 may apply. In particular, INSERT_2 skips over any list elements whose Lamport timestamp is greater than k_{new}. However, we know that $k_2 < k_{new} \vee k_2 = \mathsf{tail}$, and so INSERT_1 will apply with $next = k_2$ at the latest. The $\text{INSERT}_{1,2}$ rules thus never iterate past k_2, and thus k_{new} is never inserted at a list position that appears after k_2. □

Definition 5 (common ancestor)**.** *In a history H, the* common ancestor *of two concurrent operations o_r and o_s is the latest document state that causally precedes both o_r and o_s.*

The common ancestor of o_r and o_s can be defined more formally as the document state resulting from applying a sequence of operations $\langle o_1, \ldots, o_j \rangle$ that is the shortest prefix of H that satisfies $(o_r.deps \cap o_s.deps) \subseteq \{o_1.id, \ldots, o_j.id\}$.

Definition 6 (insertion interval)**.** *Given two concurrent operations o_r and o_s that insert into the same list, the* insertion interval *of o_r is the pair of keys $(k_r^{\text{before}}, k_r^{\text{after}})$ such that $o_r.id$ appears after k_r^{before} when o_r has been applied, k_r^{after} appears after $o_r.id$ when o_r has been applied, and k_r^{after} appears immediately after k_r^{before} in the common ancestor of o_r and o_s. The insertion interval of o_s is the pair of keys $(k_s^{\text{before}}, k_s^{\text{after}})$ defined similarly.*

It may be the case that k_r^{before} or k_s^{before} is head, and that k_r^{after} or k_s^{after} is tail.

Lemma 4. *For any two concurrent insertion operations o_r, o_s in a history H, if $o_r.cur = o_s.cur$, then the order at which the inserted elements appear in the list after applying H is deterministic and independent of the order of o_r and o_s in H.*

Proof. Without loss of generality, assume that $o_r.id < o_s.id$ according to the ordering relation on Lamport timestamps. (If the operation ID of o_r is greater than that of o_s, the two operations can be swapped in this proof.) We now distinguish the two possible orders of applying the operations:

1) o_r is applied before o_s in H. Thus, at the time when o_s is applied, o_r has already been applied. When applying o_s, since o_r has a lesser operation ID, the rule INSERT$_1$ applies with $next = o_r.id$ at the latest, so the insertion position of o_s must appear before o_r. It is not possible for INSERT$_2$ to skip past o_r.
2) o_s is applied before o_r in H. Thus, at the time when o_r is applied, o_s has already been applied. When applying o_r, the rule INSERT$_2$ applies with $next = o_s.id$, so the rule skips past o_s and inserts o_r at a position after o_s. Moreover, any list elements that appear between $o_s.cur$ and o_s at the time of inserting o_r must have a Lamport timestamp greater than $o_s.id$, so INSERT$_2$ also skips over those list elements when inserting o_r. Thus, the insertion position of o_r must be after o_s.

Thus, the insertion position of o_r appears after the insertion position of o_s, regardless of the order in which the two operations are applied. The ordering depends only on the operation IDs, and since these IDs are fixed at the time the operations are generated, the list order is determined by the IDs. □

Lemma 5. *In an operation history H, an insertion operation is commutative with respect to concurrent insertion operations to the same list.*

Proof. Given any two concurrent insertion operations o_r, o_s in H, we must show that the document state does not depend on the order in which o_r and o_s are applied.

Either o_r and o_s have the same insertion interval as defined in Definition 6, or they have different insertion intervals. If the insertion intervals are different, then by Lemma 3 the operations cannot affect each other, and so they have the same effect regardless of their order. So we need only analyze the case in which they have the same insertion interval $(k^{\text{before}}, k^{\text{after}})$.

If $o_r.cur = o_s.cur$, then by Lemma 4, the operation with the greater operation ID appears first in the list, regardless of the order in which the operations are applied. If $o_r.cur \neq o_s.cur$, then one or both of the cursors must refer to a list element that appears between k^{before} and k^{after}, and that did not yet exist in the common ancestor (Definition 5).

Take a cursor that differs from k^{before}: the list element it refers to was inserted by a prior operation, whose cursor in turn refers to another prior operation, and so on. Following this chain of cursors for a finite number of steps leads to an operation o_{first} whose cursor refers to k^{before} (since an insertion operation always inserts at a position after the cursor).

Note that all of the operations in this chain are causally dependent on o_{first}, and so they must have a Lamport timestamp greater than o_{first}. Thus, we can apply the same argument as in Lemma 4: if INSERT$_2$ skips over the list element inserted by o_{first}, it will also skip over all of the list elements that are causally dependent on it; if INSERT$_1$ inserts a new element before o_{first}, it is also inserted before the chain of operations that is based on it.

Therefore, the order of o_r and o_s in the final list is determined by the Lamport timestamps of the first insertions into the insertion interval after their common ancestor, in the chains of cursor references of the two operations. Since the argument above applies to all pairs of concurrent operations o_r, o_s in H, we deduce that the final order of elements in the list depends only on the operation IDs but not the order of application, which shows that concurrent insertions to the same list are commutative. □

Lemma 6. *In a history H, a deletion operation is commutative with respect to concurrent operations.*

Proof. Given a deletion operation o_d and any other concurrent operation o_c, we must show that the document state after applying both operations does not depend on the order in which o_d and o_c were applied.

The rules in Figure 12 define how a deletion operation o_d is applied: starting at the cursor in the operation, they recursively descend the subtree, removing $o_d.deps$ from the presence set pres(k) at all branch nodes in the subtree, and updating all registers to remove any values written by operations in $o_d.deps$.

If o_c is an assignment or insertion operation, the ASSIGN rule adds $o_c.id$ to the mapping from operation ID to value for a register, and the DESCEND, ASSIGN, EMPTY-MAP and EMPTY-LIST rules add $o_c.id$ to the presence sets pres(k) along the path through the document tree described by the cursor.

If $o_d.cur$ is not a prefix of $o_c.cur$, the operations affect disjoint subtrees of the document, and so they are trivially commutative. Any state changes by DESCEND and ADD-ID$_1$ along the shared part of the cursor path are applied using the set union operator $\cup$, which is commutative.

Now consider the case where $o_d.cur$ is a prefix of $o_c.cur$. Since o_c is concurrent with o_d, we know that $o_c.id \notin o_d.deps$. Therefore, if o_c is applied before o_d in the history, the CLEAR-* rules evaluating o_d will leave any occurrences of $o_c.id$ in the document state undisturbed, while removing any occurrences of operations in $o_d.deps$.

If o_d is applied before o_c, the effect on presence sets and registers is the same as if they had been applied in the reverse order. Moreover, o_c applies in the same way as if o_d had not been applied previously, because applying a deletion only modifies presence sets and registers, without

actually removing map keys or list elements, and because the rules for applying an operation are not conditional on the previous content of presence sets and registers.

Thus, the effect of applying o_c before o_d is the same as applying o_d before o_c, so the operations commute. □

Lemma 7. *In a history H, an assignment operation is commutative with respect to concurrent operations.*

Proof. Given an assignment o_a and any other concurrent operation o_c, we must show that the document state after applying both operations does not depend on the order in which o_a and o_c were applied.

The rules ASSIGN, EMPTY-MAP and EMPTY-LIST define how an assignment operation o_a is applied, depending on the value being assigned. All three rules first clear any causally prior state from the cursor at which the assignment is occurring; by Lemma 6, this clearing operation is commutative with concurrent operations, and leaves updates by concurrent operations untouched.

The rules also add $o_a.id$ to the presence set identified by the cursor, and DESCEND adds $o_a.id$ to the presence sets on the path from the root of the document tree described by the cursor. These state changes are applied using the set union operator $\cup$, which is commutative.

Finally, in the case where value assigned by o_a is a primitive and the ASSIGN rule applies, the mapping from operation ID to value is added to the register by the expression $child[\,id \mapsto val\,]$. If o_c is not an assignment operation or if $o_a.cursor \neq o_c.cursor$, the operations are independent and thus trivially commutative.

If o_a and o_c are assignments to the same cursor, we use the commutativity of updates to a partial function: $child[\,id_1 \mapsto val_1\,][\,id_2 \mapsto val_2\,] = child[\,id_2 \mapsto val_2\,][\,id_1 \mapsto val_1\,]$ provided that $id_1 \neq id_2$. Since operation IDs (Lamport timestamps) are unique, two concurrent assignments add two different keys to the mapping, and their order is immaterial.

Thus, all parts of the process of applying o_a have the same effect on the document state, regardless of whether o_c is applied before or after o_a, so the operations commute. □

Lemma 8. *Given an operation history $H = \langle o_1, \ldots, o_n \rangle$ from a valid execution, a new operation o_{n+1} from that execution can be inserted at any point in H after $o_{n+1}.deps$ have been applied. For all histories H' that can be constructed this way, the document state resulting from applying the operations in H' in order is the same, and independent of the ordering of any concurrent operations in H.*

Proof. H can be split into a prefix and a suffix, as described in the proof of Theorem 1. The suffix contains only operations that are concurrent with o_{n+1}, and we allow o_{n+1} to be inserted at any point after the prefix. We then prove the lemma case-by-case, depending on the type of mutation in o_{n+1}.

If o_{n+1} is a deletion, by Lemma 6 it is commutative with all operations in the suffix, and so o_{n+1} can be inserted at any point within, before, or after the suffix without changing its effect on the final document state. Similarly, if o_{n+1} is an assignment, by Lemma 7 it is commutative with all operations in the suffix.

If o_{n+1} is an insertion, let o_c be any operation in the suffix, and consider the cases of o_{n+1} being inserted before and after o_c in the history. If o_c is a deletion or assignment, it is commutative with o_{n+1} by Lemma 6 or Lemma 7 respectively. If o_c is an insertion into the same list as o_{n+1}, then by Lemma 5 the operations are commutative. If o_c is an insertion into a different list in the document, its effect is independent from o_{n+1} and so the two operations can be applied in any order.

Thus, o_{n+1} is commutative with respect to any concurrent operation in H. Therefore, o_{n+1} can be inserted into H at any point after its causal dependencies, and the effect on the final document state is independent of the position at which the operation is inserted. □

This completes the induction step in the proof of Theorem 1, and thus proves convergence of our datatype.