Specification and Complexity of Collaborative Text Editing

Hagit Attiya  
Technion  
Adam Morrison  
Technion

Sebastian Burckhardt  
Microsoft Research  
Hongseok Yang  
University of Oxford

Alexey Gotsman  
IMDEA Software Institute  
Marek Zawirski  
Inria & Sorbonne Universités, UPMC Univ Paris 06, LIP6

ABSTRACT

Collaborative text editing systems allow users to concurrently edit a shared document, inserting and deleting elements (e.g., characters or lines). There are a number of protocols for collaborative text editing, but so far there has been no precise specification of their desired behavior, and several of these protocols have been shown not to satisfy even basic expectations. This paper provides a precise specification of a replicated list object, which models the core functionality of replicated systems for collaborative text editing. We define a strong list specification, which we prove is implemented by an existing protocol, as well as a weak list specification, which admits additional protocol behaviors.

A major factor determining the efficiency and practical feasibility of a collaborative text editing protocol is the space overhead of the metadata that the protocol must maintain to ensure correctness. We show that for a large class of list protocols, implementing either the strong or the weak list specification requires a metadata overhead that is at least linear in the number of elements deleted from the list. The class of protocols to which this lower bound applies includes all list protocols that we are aware of, and we show that one of these protocols almost matches the bound.

Keywords

Collaborative text editing; eventual consistency

1. INTRODUCTION

Collaborative text editing systems, like Google Docs [6, 7], Apache Wave [1], or wikis [15], allow users at multiple sites to concurrently edit the same document. To achieve high responsiveness and availability, such systems often replicate the document in geographically distributed sites or on user devices. A user can modify the document at a nearby replica, which propagates the modifications to other replicas asynchronously. This propagation can be done either via a centralized server or peer-to-peer. An essential feature of a collaborative editing system is that all changes eventually propagate to all replicas and get incorporated into the document in a consistent way. In particular, such systems aim to guarantee eventual consistency: if users stop modifying the document, then the replicas will eventually converge to the same state [28, 29].

Figure 1(a) gives an example scenario of a document edited at several replicas. First, replica $R_2$ inserts $x$ at the first position (zero-indexed) into the empty list. This insertion then propagates to replica $R_1$, which inserts $a$ to the left of $x$, and to $R_3$, which inserts $b$ to the right of $x$. Later the modifications made by $R_1$ and $R_3$ propagate to all other replicas, including $R_2$; when the latter reads the list, it observes $axb$. In this scenario, the desired system behavior is straightforward, but sometimes this is not the case. To illustrate, consider the scenario in Figure 1(b), where $R_2$ deletes $x$ from the list before the insertions of $a$ and $b$ propagate to it. One might expect the read by $R_3$ to return $ab$, given the orderings $ax$ and $zb$ established at other replicas. However, some implementations allow $ba$ as a response; e.g., this is the case in a Jupiter protocol [19], used in public collaboration systems [30].

There have been a number of proposals of highly available collaborative editing protocols, using techniques such as operation transformations [12, 22, 26, 27] and replicated data types (aka CRDTs) [21, 23, 31]. However, specifications of their desired behavior [16, 27] have so far been informal and imprecise, and several of the protocols have been shown not to satisfy even the basic expectation of eventual consistency [13]. To address this problem, we introduce a precise specification of a replicated list object, which allows its clients to insert and delete elements into the list at different replicas and thereby captures the core aspects of collaborative text editing [12] (Section 3). Our specification has two flavors. The strong specification ensures that orderings relative to deleted elements hold even after the deletion, thereby disallowing the response $ba$ for the read in Figure 1(b). The weak specification provides no such guarantee, while still requiring the ordering between elements.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PODC ’16, July 25–28, 2016, Chicago, IL, USA.

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-3964-3/16/07...
$15.00
DOI: http://dx.doi.org/10.1145/2933057.2933090
that are not deleted to be consistent across the system. We show that both of these specifications ensure eventual consistency.

We prove that the strong specification is correctly implemented by a variant of an existing RGA (Replicated Growable Array) protocol [23], which is in the style of replicated data types [21] (Section 4). The protocol represents the list as a tree, with read operations traversing the tree in a deterministic order. Inserting an element $x$ right after an element $z$ (as in Figure 1(b)) adds $x$ as a child of $z$ in the tree. Deleting an element $x$ just marks it as such; the node of $x$ is left in the tree, creating a so-called tombstone. Keeping the tombstone enables the protocol to correctly incorporate insertions of elements received from other replicas that are ordered right after $x$ (e.g., that of $b$ in Figure 1(b)).

The simplicity of handling deletions via tombstones in the RGA protocol comes with a high space overhead. More precisely, the metadata overhead [4] of a list implementation is the ratio between the size of a replica’s state (in bits) and the size of the user-observable content of the state, i.e., the list that will be read in this state. As we show, the metadata overhead of the RGA protocol is $O(D \log k)$, where $D$ is the number of deletions issued by clients and $k$ is the total number of operations (Section 4). The number of deletions can be high. For example, a 2009 study [31] indicates that the “George W. Bush” Wikipedia page has about 500 lines. However, since modifications are usually handled as deleting the original line and then inserting the revised line, the page had accumulated about 1.6 million deletions.\footnote{Wikipedia stores this information also to track the document’s edit history.}

Our main result is that this overhead is, in some sense, inherent. We prove that any protocol from a certain class which implements the list specification for $n \geq 3$ replicas inverts a metadata overhead of $\Omega(D)$, where $D$ is the number of deletions. This result holds even for the weak list specification and even if the network guarantees causal atomic broadcast [9]. The result holds for all push-based protocols, where each replica propagates list updates to its peers as soon as possible, and merges remote updates into its state as soon as they arrive (we give a precise definition in Section 5). This assumption captures the operation of all highly available protocols that we are aware of.

We establish our lower bound for the peer-to-peer model. However, using the fact that it holds for a network with causal atomic broadcast, we extend it to show that, in a push-based client/server list protocol, the metadata overhead at the clients is still $\Omega(D)$.

We prove our lower bound using an information-theoretic argument. For every $d \approx D/2$-bit string $w$, we construct a particular execution $\alpha_w$ of the protocol such that, at its end, the user-observable state $\sigma_w$ of some replica is a list of size $O(1)$ bits. We then show that, given $\sigma_w$, we can decode $w$ by exercising the protocol in a black-box manner. This implies that all states $\sigma_w$ must be distinct and, since there are $2^d$ of them, one of these states must take at least $d$ bits. The procedure that decodes $w$ from $\sigma_w$ is nontrivial and represents the key insight of our proof. It recovers $w$ one bit at a time using a “feedback loop” between two processes: one performs a black-box experiment on the protocol to recover the next bit of $w$, and the other reconstructs the corresponding steps of the execution $\alpha_w$; the messages sent in the reconstructed part of $\alpha_w$ then form the basis for the experiment to decode the next bit of $w$.

2. SYSTEM MODEL

We are concerned with highly available implementations of a replicated object [2, 4], which supports a set of operations $Op$. Such an implementation consists of replicas that receive and respond to user operations on the object and use message passing to communicate changes to the object’s state. The high availability property sets this model apart from standard message-passing models: we require that replicas respond to user operations immediately—without performing any communication—so that user operations complete regardless of network latency and network partitions (e.g., device disconnection).

**Replicas.** We model a replica as a state machine $R = (Q, M, \Sigma, \sigma_0, E, \Delta)$, where $Q$ is a set of internal states, $M$ is a set of possible messages, $\Sigma = Q \times (M \cup \{\bot\})$ is a set of replica states, $\sigma_0 = (q_0, \bot) \in \Sigma$ is the initial state, $E$ is a set of possible events, and $\Delta: \Sigma \times E \rightarrow \Sigma$ is a (partial) transition function. Note that a replica state explicitly includes a send buffer, containing the message pending transmission or $\bot$, which indicates that no message is pending. If $\Delta(\sigma, e)$ is defined, we say that event $e$ is enabled in state $\sigma$. Transitions determined by $\Delta$ describe local steps of a replica in which it interacts with users and other replicas. These interactions are modeled by three kinds of events:

- $do(op, v)$: a user invokes an operation $op \in Op$ on the replicated object and immediately receives a response $v$ from the replica;
- $send(m)$: the replica broadcasts a message $m \in M$; and
- $receive(m)$: the replica receives a message $m \in M$.

A protocol is a collection $\mathcal{R}$ of replicas.

We require that a $send(m)$ event is enabled in state $\sigma$ if and only if $\sigma = (q, m)$ for $m \neq \bot$, and in this case $\Delta((q, m), send(m)) = (q', \bot)$ for some $q'$. We further require that a replica can execute any operation with its return values computed deterministically: for any operation $op \in Op$, exactly one $do(op, v)$ event is enabled in $\sigma$. We also require that a replica can accept any message: for any message $m$, $receive(m)$ is enabled in $\sigma$. We assume that messages are unique and that a message’s sender is uniquely identifiable (e.g., messages are tagged with the sender id and a sequence number). We also assume that a replica broadcasts messages to all replicas, including itself\footnote{Wikipedia stores this information also to track the document’s edit history.}; replicas can implement point-to-point communication by ignoring messages for which they are not the intended recipient.

**Executions.** An execution of a protocol $\mathcal{R}$ is (a possibly infinite) sequence of events occurring at the replicas in $\mathcal{R}\footnote{\text{The latter is used to support atomic broadcast [9], defined later.}}$. For each event $e$, we let $rep(e) \in \mathcal{R}$ be the replica at which it occurs, and for each $do$ event $e = do(op, v)$ we let $op(e) = op$ and $val(e) = v$. A (finite or infinite) sequence of events $e_1, e_2, \ldots$ occurring at a replica $R = (Q, M, \Sigma, \sigma_0, E, \Delta)$ is well-formed if there is a sequence of states $\sigma_1, \sigma_2, \ldots$ such that $\sigma_i = \Delta(\sigma_{i-1}, e_i)$ for all $i$. If the sequence is of length $n$, we refer to $\sigma_n$ as the state of $R$ at the end of the sequence.

We consider only well-formed executions, in which for every replica $R \in \mathcal{R}$: (1) the subsequence of events at $R$, denoted $\alpha_R$, is well-formed; and (2) every $receive(m)$ event at $R$ is preceded by a $send(m)$ event in $\alpha$.

Let $\alpha$ be an execution. Event $e \in \alpha$ happens before event $e' \in \alpha$ [14] (written $e \leadsto_{hb} e'$, or simply $e \leadsto e'$ if the context is clear) if one of the following conditions holds: (1) Thread of execution: $rep(e) = rep(e')$ and $e$ precedes $e'$ in $\alpha$. (2) Message delivery: $e = send(m)$ and $e' = receive(m)$. (3) Transitivity: There is an event $f \in \alpha$ such that $e \leadsto f$ and $f \leadsto e'$.\footnote{Formally, an execution consists of events instrumented with unique event ids and replicas. In the paper we do not use this more accurate formulation so as to avoid clutter.}
Network model. To ensure that every operation eventually propagates to all the replicas, we require that the network does not remain partitioned indefinitely. A replica \( R \) has a message pending in event \( e \) of execution \( \alpha \) if \( R' \)’s has a send \((m)\) event enabled in the state at the end of \( \alpha'\mid R \), where \( \alpha' \) is the prefix of \( \alpha \) ending with \( e \).

**Definition 1.** The network is sufficiently connected in an infinite well-formed execution \( \alpha \) of a protocol \( R \) if the following conditions hold for all replicas \( R \in \mathcal{R} \): (1) Eventual transmission: if \( R \) has a message pending infinitely often in \( \alpha \), then \( R \) also sends a message infinitely often in \( \alpha \), and (2) Eventual delivery: if \( R \) sends a message \( m \), then every replica \( R' \neq R \) eventually receives \( m \).

Collaborative editing protocols generally assume causal message delivery \([23, 26]\). We model this by considering only executions that satisfy causal broadcast \([5]\):

**Definition 2.** An execution \( \alpha \) of a protocol \( \mathcal{R} \) satisfies causal broadcast if for any messages \( m, m' \), whenever \( \text{send}(m) \overset{\text{vis}}{\rightarrow} \text{send}(m') \), any replica \( R \) can receive \( m' \) only after it receives \( m \).

In fact, our results hold even under a more powerful atomic broadcast \([9]\) model, which delivers all messages to all replicas in the exact same order.

**Definition 3.** An execution \( \alpha \) of protocol \( \mathcal{R} \) satisfies causal atomic broadcast if the following conditions hold: (1) Causal broadcast: \( \alpha \) satisfies causal broadcast. (2) No duplicate delivery: each send \((m)\) event in \( \alpha \) is followed by at most one receive \((m)\) event per replica \( R' \in \mathcal{R} \). (3) Consistent order: if \( R \) receives \( m \) before \( m' \), then any other replica \( R' \) receives \( m \) before \( m' \).

These broadcast primitives can be implemented when not provided by the network \([5]\); by providing them “for free,” we strengthen our lower bounds and ensure their independence from the complexity of implementing the broadcast primitive.

### 3. COLLABORATIVE TEXT EDITING

Following Ellis and Gibbs \([12]\), we model the collaborative text editing problem (henceforth, simply collaborative editing) as the problem of implementing a highly available replicated list object whose elements are from some universe \( U \). Users can insert elements, remove elements and read the list using the following operations, which form Op:

- \( \text{ins}(a, k) \) for \( a \in U \) and \( k \in \mathbb{N} \): inserts \( a \) at position \( k \) in the list (starting from 0) and returns the updated list. For \( k \) exceeding the list size, we assume an insertion at the end. We assume that users pass identifiers \( a \) that are globally unique.
- \( \text{del}(a) \) for \( a \in U \): deletes the element \( a \) and returns the updated list. We assume that users pass only identifiers \( a \) that appear in the return value of the preceding operation on the same replica.
- read: returns the contents of the list.

The definition above restricts user behavior to simplify our technical development. Note that these restrictions are insignificant from a practical viewpoint, because they can be easily enforced: (1) identifiers can be made unique by attaching replica identifiers and sequence numbers; and (2) before each deletion, we can read the state of the list and skip the deletion if the deleted element does not appear in it.

#### 3.1 Preliminaries: Replicated Data Types

We cannot specify the list object with a standard sequential specification, since replicas may observe only subsets of operations executed in the system, as a result of remote updates being delayed by the network. We address this difficulty by specifying the response of a list operation based on operations that are visible to it. Intuitively, these are the prior operations executed at the same replica and remote operations whose effects have propagated to the replica through the network. Formally, we use a variant of a framework by Burckhardt et al. \([4]\) for specifying replicated data types \([25]\). We specify the list object by a set of abstract executions, which record the operations performed by users (represented by \( do \) events) and visibility relationships between them. Since collaborative editing systems generally preserve causality between operations \([26]\), here we consider only causal abstract executions, where the visibility relation is transitive.

**Definition 4.** A causal abstract execution is a pair \((H, \text{vis})\), where \( H \) is a sequence of \( do \) events, and \( \text{vis} \subseteq H \times H \) is an acyclic visibility relation (with \((e_1, e_2) \in \text{vis} \) denoted by \( e_1 \xrightarrow{\text{vis}} e_2 \)) such that: (1) if \( e_1 \) precedes \( e_2 \) in \( H \) and \( \text{repl}(e_1) = \text{repl}(e_2) \), then \( e_1 \xrightarrow{\text{vis}} e_2 \); (2) if \( e_1 \xrightarrow{\text{vis}} e_2 \), then \( e_1 \) precedes \( e_2 \) in \( H \); and (3) \( \text{vis} \) is transitive (if \( e_1 \xrightarrow{\text{vis}} e_2 \) and \( e_2 \xrightarrow{\text{vis}} e_3 \), then \( e_1 \xrightarrow{\text{vis}} e_3 \)).

Figure 1 graphically depicts abstract executions, where \( \text{vis} \) is the transitive closure of arrows in the figure and \( H \) is the result of some topological sort of \( \text{vis} \). An abstract execution \( \alpha' = (H', \text{vis}') \) is a prefix of abstract execution \( \alpha \) if: (1) \( H' \) is a prefix of \( H \); and (2) \( \text{vis}' = \text{vis} \cap (H' \times H') \). A specification of an object is a pre-fixed closed set of abstract executions. A protocol correctly implements a specification when the outcomes of operations that it produces in any (concrete) execution can be justified by some abstract execution allowed by the specification.

**Definition 5.** An execution \( \alpha \) of a protocol \( \mathcal{R} \) complies with an abstract execution \( \alpha = (H, \text{vis}) \) if for every replica \( R \in \mathcal{R} \), \( H_{\mid R} = \alpha_{\mid R} \), where \( \alpha_{\mid R} \) denotes the subsequence of \( do \) events by replica \( R \) in \( \alpha \).

**Definition 6.** A protocol \( \mathcal{R} \) satisfies a specification \( \mathcal{S} \) if every execution \( \alpha \) of \( \mathcal{R} \) complies with some abstract execution \( \alpha \in \mathcal{S} \).

#### 3.2 Specifying the List Object

We present two list specifications: strong and weak. Conceptually, the strong specification ensures that orderings relative to deleted elements hold even after the deletion, thereby disallowing the response \( ba \) for the read in Figure 1(b). The weak specification does not guarantee this property, allowing both \( ba \) and \( ab \) as responses.

We denote by \( \text{elems}(A) \) the set of all elements inserted into the list in an abstract execution \( A = (H, \text{vis}) \):

\[
\text{elems}(A) = \{ e \mid \text{do}(\text{ins}(a, \_)) \in H \}.
\]

Recall that we assume all inserted elements to be unique, and so there is a one-to-one correspondence between inserted elements and insert operations. For brevity, we write \( e_1 \leq_{\text{vis}} e_2 \) for \( e_1 = e_2 \lor e_1 \xrightarrow{\text{vis}} e_2 \).

**Definition 7.** An abstract execution \( A = (H, \text{vis}) \) belongs to the strong list specification \( A_{\text{strong}} \) if and only if there is a relation \( \leq \subseteq \text{elems}(A) \times \text{elems}(A) \), called the list order, such that:

\[\text{Formally, } H \text{ consists of } do \text{ events instrumented with unique event ids and replicas, as in the case of an execution } \alpha. \text{ To avoid clutter, we do not use this more accurate presentation.}\]
1. Each event e = do(op, w) ∈ H returns a sequence of elements w = a₀ ⋯ aₙ₋₁, where aᵢ ∈ elems(A), such that
   (a) w contains exactly the elements visible to e that have been inserted, but not deleted:
   \[ \forall a. a ∈ w \iff (\text{do}(\text{ins}(a, \_)), \_) \leq_{vis} e \land \neg(\text{do}(\text{del}(a, \_)), \_) \leq_{vis} e. \]
   (b) The order of the elements is consistent with the list order:
   \[ \forall i, j. (i < j) \implies (aᵢ, aⱼ) ∈ \text{lo}. \]
   (c) Elements are inserted at the specified position: if \( op = \text{ins}(a, k) \), then \( a = a_{\text{min}}(k, n-1). \)

2. The list order \( \text{lo} \) is transitive, irreflexive and total, and thus determines the order of all insert operations in the execution.

For example, the strong list specification is satisfied by the abstract execution in Figure 1(a) and the one in Figure 1(b) with the read returning \( ab \); this is justified by the list order \( a \rightarrow x \rightarrow b \). On the other hand, the specification is not satisfied by the execution in Figure 1(b) with the read returning \( ba \): for the outcomes of operations in this execution to be consistent with item 1 of Definition 7, the list order would have to be as shown above; but this order contains a cycle, contradicting item 2. In Section 4 we prove that the strong specification is implemented by an existing protocol, RGA [23]. However, some protocols, such as Jupiter [19], provide weaker guarantees and, in particular, allow the outcome \( ba \) in Figure 1(b). We therefore introduce the following weak list specification, to which our lower bound result applies (Section 6)³.

DEFINITION 8. An abstract execution \( A = (H, \text{vis}) \) belongs to the weak list specification \( A\text{weak} \) if and only if there exists a relation \( \text{lo} ⊆ \text{elems}(A) × \text{elems}(A) \) such that:

1. Condition 1 in Definition 7 is satisfied.
2. \( \text{lo} \) is irreflexive and, for all events \( e = \text{do}(op, w) \in H \), it is transitive and total on \( \{a \mid a \in w\} \).

Unlike the strong specification, the weak one allows the list order \( \text{lo} \) to have cycles; the order is required to be acyclic only on the elements returned by some operation. In particular, the weak specification allows the execution in Figure 1(b) with the read returning \( ba \), which is justified using the above cyclic list order. Since at the time of the read, \( x \) is deleted from the list, the specification permits us to decide how to order \( a \) and \( b \) without taking into account the orderings involving \( x : a \rightarrow x \) and \( x \rightarrow b \).

Eventual consistency. A desirable property of highly available replicated objects is eventual consistency. Informally, this guarantees that, if users stop issuing update requests, then the replicas will eventually converge to the same state [28, 29]. Our specifications imply a related convergence property: in an abstract execution satisfying \( A\text{strong} \) or \( A\text{weak} \), two read operations that see the same sets of list updates return the same response. This is because such operations will return the same elements (Definition 7, item 1a) and in the same order (Definition 7, item 1b). From the convergence property we can establish that our specifications imply eventual consistency for a class of protocols that guarantee the following property of eventual visibility.

DEFINITION 9. An abstract execution \( A = (H, \text{vis}) \) satisfies eventual visibility if for every event \( e \in H \), there are only finitely many events \( e′ \in H \) such that \( \neg(e \leq_{vis} e′).\)

³We conjecture that Jupiter satisfies the weak specification.

DEFINITION 10. A protocol \( R \) satisfying the weak (resp., strong) list specification guarantees eventual visibility if every execution \( \alpha \) of \( R \) complies with some abstract execution \( A \in A\text{weak} \) (respectively, \( A \in A\text{strong} \)) that satisfies eventual visibility.

Informally, eventual consistency holds for a protocol guaranteeing eventual visibility because: in an abstract execution with finitely many list updates, eventual visibility ensures that all but finitely many reads will see all the updates; then convergence ensures that they will return the same list. To guarantee eventual visibility, a protocol would rely on the network being sufficiently connected (Definition 1).

3.3 Metadata Overhead

In addition to the user-observable list contents, the replica state in a list protocol typically contains user-unobservable metadata that is used internally to provide correct behavior. The metadata overhead is the proportion of metadata relative to the user-observable list content.

Formally, let the size of an internal replica state \( q \) or a list \( w \in U \) be the number of bits required to represent it in a standard encoding; we denote the size of \( x \) by \( |x| \). The metadata overhead [4] of a state \( σ = (q, m) \) is \( |q|/|w| \) for the unique \( w \) such that \( \text{do}(read, w) \) is enabled in \( σ \); here \( w \) represents the user-observable contents of \( σ \). Note that the contents of the send buffer is not part of the metadata.

DEFINITION 11 ([4]). The worst-case metadata overhead of a protocol over a given subset of its executions is the largest metadata overhead of the state of any replica in any of these executions.

4. AN IMPLEMENTATION OF THE STRONG LIST SPECIFICATION

We now present an implementation of the list object, which is a reformulation of the RGA (Replicated Growable Array) protocol [23], and prove that it implements the strong list specification.

4.1 Timestamped Insertion Trees

Our representation of the list at a replica uses a timestamped insertion (TI) tree data structure. It stores both the list content and timestamp metadata used for deterministically resolving the order between elements concurrently inserted at the same position.

Formally, a tree is a finite set \( N \) of nodes, each corresponding to an element inserted into the list. A node is a tuple \( n = (a, t, p) \), where \( a ∈ U \) is the element, \( t ∈ C \) is a timestamp for the insertion, and \( p ∈ (C \cup \{\circ\}) \) is either the parent node (identified by its timestamp) or the symbol \( \circ \) representing the tree root. We define the set \( C \) of timestamps and a total order on them later (Section 4.2). For a node \( n = (a, t, p) \) we let \( n.a = a, n.t = t \), and \( n.p = p \). For two nodes \( n, n′ \) with \( n’.p = n.t \), we say \( n \) is the parent of \( n′ \) and write \( n ↘ n′ \).

DEFINITION 12. A set of nodes \( N \) is a TI tree if (1) timestamps uniquely identify nodes: \( ∀n, n′ \in N : n.t = n′.t \implies n = n′; \) (2) all parents are present: if \( n ∈ N \) and \( n.p ≠ \circ \), then \( n.p ↗ n \) for some \( n′ ∈ N \); and (3) parents are older than their children: \( n ↗ n′ \implies n.t < n′.t \).

Figure 2 shows an example of a TI tree and illustrates the read and insert operations explained below.

Read. To read the list, we traverse the tree \( N \) by depth-first search, starting at the root. We assemble the visited elements into a sequence \( s(N) \) using prefix order (the parent precedes its children) and visit the children in decreasing timestamp order.
Insert. To insert a new element $a$ at position $k$ into the list, let $s(N) = a_0 \ldots a_{n-1}$ and pick a new timestamp $t$ that is larger than any of the timestamps appearing in $N$. Then let $p$ be the element to the left of the insertion position: $p = a_{k-1}$ (if $k > 0$) or $p = o$ (if $k = 0$). We now add a new node $(a, t, p)$ to $N$. Note that a newly inserted node is the child of the immediately preceding element with the highest timestamp. Thus, it is visited immediately after that element during a read, which makes it appear at the correct position in the list.

4.2 The RGA Protocol

We now define the RGA protocol $\mathcal{R}_{\text{rga}}$ for $n$ replicas. Each replica stores a TI tree, as well as a set of elements that represent tombstones, used to handle deletions (Section 1). Insertions and deletions are recorded in a send buffer, which is periodically transmitted to other replicas by causal broadcast.

State and messages. Timestamps are pairs $(x, i)$, where $x \in \mathbb{N}$ and $i \in \{1, \ldots, n\}$ is a replica identifier. They are ordered lexicographically:

$$(x, i) < (x', i') \iff (x < x') \vee ((x = x') \land (i < i')).$$

Messages are of the form $(A, K)$, where $A$ is a set of nodes (representing insert operations) and $K$ is a set of elements (representing delete operations), and either $A$ or $K$ is non-empty. The state of a replica is $(N, T, (A, K))$, where: $N$ is a TI tree, representing the replica-local view of the list; $T \subseteq U$ is the set of tombstones; and $(A, K)$ is a send buffer, containing the message to send next. A pair $(\emptyset, \emptyset)$ indicates that no message is pending (thus corresponding to $\bot$ in Section 2). The initial state is $(\emptyset, \emptyset, (\emptyset, \emptyset))$.

Do transitions. To execute an insert operation at a replica $i$ in a state $(N, T, (A, K))$, we construct a node as described in the “Insert” procedure of Section 4.1 and add it to both $N$ and $A$. As the timestamp of the node we take $((1 + \text{the largest timestamp in } N), i)$, or $(1, i)$ if $N = \emptyset$. This timestamp is guaranteed to be globally unique. To execute a delete operation, we add the deleted element to both $T$ and $K$. All operations return the local view of the list, which is obtained by traversing $N$ as described in the “Read” procedure of Section 4.1, and then removing all elements belonging to $T$.

Send transition is enabled whenever either $A$ or $K$ is non-empty. It sends $(A, K)$ as the message and sets both $A$ and $K$ to empty.

Receive transition for a message $(A_m, K_m)$ adds $A_m$ to $N$ and $K_m$ to $T$. The protocol relies on causal delivery of messages, which ensures that no parents can be missing from $N$. In particular, $N$ stays well-formed after adding $A_m$.

Figure 2: Illustration of TI trees. Each box shows a tree, with the set of nodes that define it, its graphical representation, and the sequence of elements it denotes. The tree on the right results from an insert operation for element $b$ at position 2. The order on the timestamps is $t_1 < t_2 < t_3 < t_4$.

Figure 3: Illustration of an RGA execution, with time proceeding from top to bottom. Bullets show the transitions of the replicas $R_1$, $R_2$, and $R_3$, and in some places we indicate the current state of the tree $N$ and the tombstone set $T$. The order on the timestamps is $t_1 < t_2 < t_3$.

We show an example execution in Figure 3, which matches the example in Figure 1(b) and complies with the strong list specification.

4.3 Guarantees

The following theorems state the correctness and asymptotic complexity bounds of RGA. We provide full proofs in Appendix A and discuss the key insights (convergence and stability) below.

Theorem 1. The protocol $\mathcal{R}_{\text{rga}}$ satisfies the strong list specification.

Theorem 2. The worst-case metadata overhead of $\mathcal{R}_{\text{rga}}$ over executions with $k$ operations and $D$ deletions is $O(D \log k)$.

Convergence. Each replica maintains a TI tree that grows over time, meaning that nodes are added to the set, but never modified or removed. Because set union is associative and commutative, the order in which nodes are added does not matter. For example, changing the order of message delivery to $R_2$ in Figure 3 does not change the final tree. As a consequence, if the same set of nodes is delivered to any two replicas in any order, their trees are guaranteed to match, which ensures convergence.

Stability. The following lemma (proved in Appendix A) shows that when we add more elements to a TI tree, the order of existing elements remains stable. This implies the strong list specification, because all replicas order all insertions the same way at all times.

Lemma 3. Let $A, B$ be two TI trees such that $A \subseteq B$. Then $s(A)$ is a subsequence of $s(B)$.

Trees vs. Lists. In the standard RGA implementation [24], TI trees are represented as lists (corresponding to the tree traversal). We show in Appendix A.1 that these representations are functionally equivalent. Lists are convenient to implement, but offer little insight as to why the algorithm guarantees convergence. Not surprisingly, the reason why RGA actually works has been a bit of a mystery, and we are not aware of any prior correctness proofs.

5. PUSH-BASED PROTOCOLS

Our lower bound results hold for push-based protocols, a class of protocols that contains the protocols of several collaborative editing
systems [19, 21, 23, 26], including the RGA protocol of Section 4. Informally, a replica in a push-based protocol propagates list updates to its peers as soon as possible and merges remote updates into its state as soon as they arrive (as opposed to using a more sophisticated mechanism, such as a consensus protocol). We define this class of protocols assuming that the network provides causal broadcast: when this is not the case, a protocol may need to delay merging arriving updates to enforce casuality. Formally, we require that in a push-based protocol, every operation observe all operations that happen before it, that list insertions always generate a message, and that a deletion—which, unlike an insertion, may not be unique—generates a message if it does not already observe another deletion of the same element.

**Definition 13.** A protocol \( \mathcal{R} \) satisfying the weak (strong) list specification is push-based if the following hold:

- For any execution \( \alpha \) of \( \mathcal{R} \) and \( e = \text{do}(\text{ins}(a, \_), \_) \in \alpha \), replica repl(\( e \)) has a message pending after \( e \).
- For any execution \( \alpha \) of \( \mathcal{R} \) and \( e = \text{do}(\text{del}(a, \_), \_) \in \alpha \), if there does not exist event \( e' = \text{do}(\text{del}(a, \_), \_) \in \alpha \) that happens before \( e \), then replica repl(\( e \)) has a message pending after \( e \).
- For every execution \( \alpha \) of \( \mathcal{R} \) there exists an abstract execution \( \hat{A} = (H, \text{vis}) \in \mathcal{A}_{\text{weak}}(A \in \mathcal{A}_{\text{strong}}) \) that \( \alpha \) complies with, such that \( \forall e', e \in H. \ e' \rightarrow_{\text{op}} e \iff e' \rightarrow_{\text{ins}} e \).

The class of push-based protocols contains both op-based protocols [4], in which a message carries a description of the latest operations that the sender has performed (e.g., RGA), and state-based protocols, in which a message describes all operations the sender knows about (i.e., its state). We also show (Appendix B) that the class of push-based protocols contains eventually consistent write-propagating protocols [2]—which model many deployed highly available eventually consistent protocols [3, 4, 8, 10, 11, 17, 25, 33]—under the natural assumption that sending a message does not affect the state of the list at the sending replica.

6. LOWER BOUNDS ON METADATA OVERHEAD

Here we show a lower bound on the worst-case metadata overhead (Definition 11) of a push-based protocol satisfying the weak or strong list specification.

**Theorem 4.** Let \( \mathcal{R} \) be a push-based protocol that satisfies the weak or strong list specification for \( n \geq 3 \) replicas. Then the worst-case metadata overhead of \( \mathcal{R} \) over executions with \( D \) deletions is \( \Omega(D) \).

This follows from the following theorem, because any execution consistent with the strong list specification is also consistent with the weak one.

**Theorem 5.** Let \( \mathcal{R} \) be a push-based protocol that satisfies the weak list specification for \( n \geq 3 \) replicas. Then for every integer \( D \geq 4 \), there exists an execution \( \alpha_D \) of \( \mathcal{R} \) with \( D \) deletions such that: (1) the metadata overhead of some state \( \sigma \) of some replica \( R \) in \( \alpha_D \) is \( \Omega(D) \); (2) \( \alpha_D \) satisfies causal atomic broadcast; and (3) \( \mathcal{R} \) does not receive any message before \( \sigma \) in \( \alpha_D \).

**Proof.** Let \( d = \lfloor (D - 2)/2 \rfloor \). We show that there exists an execution \( \mathcal{R} \) with \( D \) deletions that satisfies the desired conditions, in which the user-observable contents of some internal state is a list with a single element, and yet the size of this state is at least \( d \) bits. It follows that the metadata overhead of this state is \( \Omega(D) \).

We show the existence of this execution using an information-theoretic argument. Namely, for every \( d \)-bit string \( w \) we construct an execution \( \alpha_w \) that satisfies causal atomic broadcast and in which: (1) replica \( R_1 \) performs \( D \) deletions and receives no messages; (2) at the end of \( \alpha_w \), the user-observable list at \( R_1 \) contains the single element “\( w \)” and \( R_1 \) has no messages pending; and yet (3) we can decode \( w \) given only \( \sigma_w \), the state of \( R_1 \) at the end of \( \alpha_w \) (this decoding process exercises the protocol \( \mathcal{R} \) in a black-box manner). Hence, all states \( \sigma_w \) must be distinct. Since there are \( 2^d \) states, one of these states \( \sigma_w \) must take at least \( d \) bits. Since this state has no messages pending, its metadata overhead is \( \Omega(D) \), and thus, \( \alpha_w \) is the desired execution.

**Encoding \( w \).** Given a \( d \)-bit string \( w = w_1 \ldots w_d \), we construct an execution \( \alpha_w \) of \( \mathcal{R} \) that builds a list encoding the path from the root of a binary tree of height \( d \) to the \( w \)-th leaf (when \( w \) is interpreted as the binary representation of an integer). Figure 4(a) details the construction: it shows pseudocode which, as it executes, constructs the execution; instructions of the form \( \text{e}_i \) correspond to a state transition \( e_i \) at replica \( R_i \). We abuse notation by writing \( \text{op} \) instead of \( \text{do}(\text{op}, \_) \), by specifying inserts of whole strings instead of element by element, and by specifying positions relative to prior insertions rather than with integers. Figure 5(a) depicts \( \alpha_w \) for \( w = 10 \).

Only replica \( R_1 \) participates in the encoding execution \( \alpha_w \). We start by inserting the string \( \_0 [0] \) (i.e., the root). Because \( \mathcal{R} \) is a push-based protocol, \( R_1 \) has a message \( m_1 \) pending following these insertions. We then proceed with a series of steps, for \( i = 1, \ldots, d \). Each step \( i \) begins with \( R_1 \) in state \( \sigma_i \) having a message \( m_i \) pending. \( R_1 \) first broadcasts \( m_i \). We then insert the string \( [i] \), immediately to the left or to the right of \( [i-1] \), depending on whether the \( i \)-th bit of \( w \) is set. Because \( \mathcal{R} \) is a push-based protocol, \( R_1 \) has a message pending following these insertions, and we proceed to step \( i+1 \). When we are done, we broadcast the current pending message and insert the element \( * \) between \([d \text{ and } d] \), and broadcast the message \( m_{d+2} \) that is pending following this insertion. For example, if \( w = 10 \), the state of the list at \( R_1 \) at this point is \([0 \_0 [2 \_0 [1] 1] \). We then delete all the \([i] \) and \([\_i] \) elements, for \( i = 0, \ldots, d \), and if \( D \) is odd, we insert and delete an additional element, so that the number of deletions in \( \alpha_w \) is exactly \( D \). Because \( \mathcal{R} \) is a push-based protocol, \( R_1 \) has a message pending following these deletions, which we broadcast to empty \( R_1 \)'s send buffer. Finally, we read the list at \( R_1 \), observing that it is \( * \). This follows because for any abstract execution \( A = (H, \text{vis}) \) that the encoding execution \( \alpha_w \) complies with, all insert and del events are visible to the read, due to Condition (1) of Definition 4. The read's response must thus be \( * \), since by assumption one of such executions \( A \) is consistent with the weak list specification.

The output of the encoding procedure is \( \sigma_w \), the state of \( R_1 \) at the end of the encoding execution \( \alpha_w \). It is easy to check that \( \sigma_w \) is well-formed; furthermore, it vacuously satisfies causal atomic broadcast.

**Decoding \( w \) from \( \sigma_w \).** We reconstruct \( w \) one bit at a time by “replaying” the execution \( \alpha_w \). To replay iteration \( i \) of \( \alpha_w \), we rely on a procedure \( \text{Recover}() \) that recovers \( w_i \) from \( \sigma_w \) and \( m_1, \ldots, m_i \). (We describe \( \text{Recover}() \) in the next paragraph; for now, assume it is an oracle.) Knowing \( w_i \), in turn, determines the next event of \( R_1 \) in \( \alpha_w \), and hence provides us with \( m_{i+1} \). The decoding process thus only uses messages from \( R_1 \) that it reconstructs with the bits of \( w \) already known. Figure 4(b) shows the pseudocode which, as it executes, decodes \( w \). We start with \( R_1 \) in its initial state and recon-
struct $m_1$, which does not depend on $w$. We then proceed in steps, for $i = 1, \ldots, d$. In step $i$ we know $m_1, \ldots, m_i$, and we recover bit $w_i$ from $\sigma_w$ and $m_1, \ldots, m_i$. Having recovered $w_i$, we replay the insertion that $R_1$ performs at step $i$ of the encoding and reconstruct $m_{i+1}$.

Recovering $w_i$ from $\sigma_w$ and $m_1, \ldots, m_i$. The Recover() procedure determines $w_i$ by performing state transitions on fresh copies of $R_1$ and $R_2$; the transitions that an execution of Recover() performs have no effect on the state of the replica in the “replayed” execution constructed by the decoding process, or on other Recover() executions. Figure 4(c) shows these state transitions, and Figures 5(b)–5(c) illustrate the overall decoding of $w = 10$ (the use of the replica $R_2$ is explained below). We start off with $R_2$ in its initial state and $R_1$ in state $\sigma_w$. We deliver the messages $m_1, \ldots, m_i$ to both replicas in the same order. We then read at $R_2$ and receive response $v_{i-1}^w$; we will show that $[i-1] \_ - 1 \in v_{i-1}^w$. Next, $R_2$ inserts element $x \in [i-1] \_ - 1$ and broadcasts a message $m_{x-1}$, which we deliver to both replicas. Finally, we read at $R_1$ and observe the list in state $y_{i-1}^w$. We will show that $y_{i-1}^w$ contains only $x$ and $\ast$, and if $x$ precedes $\ast$ then $w_i = 1$; otherwise, $w_i = 0$.

Validity of Recover$(\sigma_w, m_1, \ldots, m_i)$ state transitions. Assuming that $m_1, \ldots, m_i$ are the first $i$ messages sent by $R_1$ in $\omega_w$, we show that the state transitions performed by an execution of Recover$(\sigma_w, m_1, \ldots, m_i)$ in Figure 4(c) occur in an extension $\beta_w'$ of $\omega_w$ of the form:

\[ \beta_w' = \omega_w \]

\[ \text{receive}_2(m_1) \text{ receive}_1(m_1) \ldots \text{receive}_2(m_i) \text{ receive}_1(m_i) \]

\[ \text{do}_2(\text{read}_2, v_{i-1}^w) \text{ do}_2(\text{ins}(x, k_0^i), \ldots) \text{ send}_2(m_x) \]

\[ \text{receive}_2(m_x) \text{ receive}_1(m_x) \text{ do}_1(\text{read}_1, y_{i-1}^w) \]

where $k_0^i$ is the position at which $R_2$ inserts $x$ into the list. In the following, we prove that the execution $\beta_w'$ is well-formed (Claim 6), that it satisfies causal atomic broadcast (Claim 7), that $[i-1] \_ - 1 \in v_{i-1}^w$ (Claim 8), and that $y_{i-1}^w = x$ or $y_{i-1}^w = \ast x$ (Claim 9). To this end, we exploit the fact that $\sigma_w$ is $R_1$’s state at the end of $\omega_w$, which allows Recover() to perform the same state transitions at $R_1$ that occur in $\beta_w'$, without having access to the entire execution $\omega_w$ that leads $R_1$ to state $\sigma_w$.

Claim 6. Execution $\beta_w'$ is well-formed.

Proof. By assumption, Recover() is passed the first $i$ messages sent by $R_1$ in $\omega_w$. The claim thus follows from the following: (1) $\omega_w$ is well-formed; (2) at the end of $\omega_w$, $R_1$ is in state $\sigma_w$; (3) because $R_2$ does not participate in $\omega_w$, the state of $R_2$ at the end of $\omega_w$ is its initial state; (4) a replica always accepts any sent message (by definition); and (5) $R$ is push-based, and so $R_2$ has a message pending following its ins operation.

Claim 7. Execution $\beta_w'$ satisfies causal atomic broadcast.

Proof. Immediate from inspection of the message delivery order in $\beta_w'$.

Claim 8. $[i-1] \_ - 1 \in v_{i-1}^w$.

Proof. For $j = 1, \ldots, d + 1$, let $e_j, f_j \in \alpha_w$ be the $do$ events in which $R_1$ inserts $[j-1]$ and $[j-1]$ into the list. Let $r \in \beta_w'$ be the $do$ event at which $R_2$ reads $v_{i-1}^w$. Because $R$ is a correct push-based protocol and $\beta_w'$ satisfies causal atomic broadcast, by Definition 13, $\beta_w'$ complies with some abstract execution $A = (H, \text{vis}) \in \mathcal{A}_{\text{weak}}$ such that $e_j \xrightarrow{\text{do}} r$ and $f_j \xrightarrow{\text{do}} r$ if and only if $j \leq i$, and no del operation is visible to $r$. This holds because in $\beta_w'$, $R_2$ receives only the messages $m_1, \ldots, m_i$ before $r$, and each $m_j$ is the first message sent by $R_1$ after $e_j$ and $f_j$. It thus follows from the definition of the weak list specification (Definition 8) that $v_{i-1}^w = \ldots [i-1] \_ - 1 \ldots$.

Claim 9. $y_{i-1}^w = x \ast$ or $y_{i-1}^w = \ast x$.

Proof. Let $f \in \beta_w'$ be the $do$ event at which $R_2$ inserts $x$ into the list, and $r \in \beta_w'$ be the $do$ event at which $R_1$ reads $y_{i-1}^w$. Because
some abstract execution $e,e_i$ events at $R_i$ from $R_i$ receives the messages generated after each insertion and then

Correctness of recovering $\alpha_w$. Having shown that the state transitions performed by Recover() yield the lists $y_w^* = x \ast$ or $y_w^* = x$, it remains to show that we correctly recover $w_i$ from $y_w^*$: $(y_w^* = x \ast) \iff (w_i = 1)$.

In principle, the weak list specification allows $R_1$’s read to order $x$ and $\ast$ arbitrarily, since $[i-1]$ and $[i-1]$ are deleted from the list by the time the read occurs. We show, however, that $R_1$ cannot do this, because it cannot rule out the possibility that another replica has already observed $[i-1]$ and $x \ast$ together, and therefore their order is fixed. Consider the following extension of $\beta_w$, in which $R_3$ receives the messages generated after each insertion and then reads the list (it is easy to see that this execution satisfies causal atomic broadcast):

$$\gamma_w^* = \beta_w^* \text{ receive}_3(m_1) \ldots \text{ receive}_3(m_i) \text{ receive}_3(m_a)$$

$$\text{ receive}_3(m_{i+1}) \ldots \text{ receive}_3(m_{d+2}) \text{ do}_3(\text{ read}_3, z_w^*).$$

We show that the list $z_w^*$ contains $\ast$ after $x$ if and only $w_i = 1$. Informally, this follows because every element inserted from iteration $i$ onwards in the encoding procedure (and hence in $\gamma_w^*$), including $\ast$, goes after $[i-1]$ if and only if $w_i = 1$, and no del events are visible to $R_3$, so its read response must order $x$ before $[i-1]$ before $\ast$.

Formally, consider the following events in $\gamma_w^*$: $w_s$, the ins of $\ast$ by $R_1$; $w_x$, the ins of $x$ by $R_2$; and $r_x$, the read by $R_2$, whose response is $z_w^*$. Because $R$ is a correct push-based protocol and $\gamma_w^*$ satisfies causal broadcast, by Definition 13, $\gamma_w^*$ complies with some abstract execution $A = (H, \text{ vis}) \in \mathcal{A}_{\text{weak}}$ such that for any $e, e' \in H$, $e' \xrightarrow{\text{ vis}} e$ if and only if $e' \xrightarrow{\text{ bb}} e$. Therefore, no del event is visible to $w_s$, $w_x$ or $r_x$. Let $lo$ be a list order that $A$ is consistent with (Definition 8). We proceed to show that $(x, \ast) \in lo$ if and only if $w_i = 1$. Observe that if $w_i = 1$, every element inserted from iteration $i$ onwards of the encoding process is inserted after $[i-1]$ and, if $w_i = 0$, every element inserted from iteration $i$ onwards is inserted before $[i-1]$. Therefore, the response of $w_i$ establishes that $([i-1], \ast) \in lo$ if and only if $w_i = 1$. The response of $w_i$ establishes that $([i-1], x) \in lo$ and $(x, [i-1]) \in lo$. It follows that $(x, \ast) \in lo$ if and only if $w_i = 1$, since $[i-1], x, [i-1], \ast \in z_w^*$ and so is total and transitive on $\{ a \mid a \in \text{ rval}(r_x) = z_w^* \}$.

We conclude by noting that $y_w^* = x \ast$ or $y_w^* = x \ast$ (Claim 9); recall that $y_w^*$ is the response to the read at $R_1$ performed by Recover(). Since $(x, \ast) \in lo$ if and only if $w_i = 1$, then $y_w^* = x \ast$ if and only if $w_i = 1$.

6.1 Extension to a Client/Server Model

In a client/server protocol, replicas communicate only with a central server and not directly with each other. (The motivation is to maintain state on the server instead of on the replicas, and so the server usually does more than merely relay messages between replicas [19].) To model such protocols in our framework, which assumes a broadcast transport, we require replicas to process only messages to/from the server:

**Definition 14.** A protocol $\mathcal{R} = \{ R_1, \ldots, R_n, S \}$ is a client/server protocol if for every replica $R_i = (Q^*, M, \Sigma^i, \sigma_0^i, E, \Delta^i)$, and $\sigma \in \Sigma^i$, if $\Delta^i(\sigma, \text{ receive}(m)) \neq \sigma$, then $m$ was sent by $S$. We call $S$ the server.

In practice, users do not interact directly with the server, and so we consider only executions in which do events do not occur at the server.

Assuming atomic broadcast, a broadcast protocol can simulate a client/server protocol using state machine replication [14].

**Proposition 10.** Let $\mathcal{R} = \{ R_1, \ldots, R_n, S \}$ be a client/server protocol. Then there exists a protocol $\mathcal{R}' = \{ R_1', \ldots, R_n' \}$
That simulates \( R \) in the following sense: (1) for any execution \( e' \) of \( R' \) that satisfies causal atomic broadcast, there exists an execution \( e \) of \( R \) such that \( o_{R'}(e') = o_R(e) \) for \( i = 1..n \); (2) the set of internal states of each \( R_i' \) is \( Q^i' \times Q^i_S \), where \( Q^i \) and \( Q^i_S \) are respective sets of \( R_i \) and \( S \); and (3) until \( R_i' \) receives a message, its state is \( (q_i^0, q_i^s) \), where \( q_i^0 \) is the initial state of \( R_i \).

**Proof.** For \( 1 = 1..n \), replica \( R_i' \in R' \) maintains two state machines, of \( R_i \) and of \( S \). \( R_i' \) broadcasts exactly the messages broadcast by the replica \( R_i \) it is simulating. We use the fact that messages are delivered to all replicas in \( R' \) in the same order to simulate the server \( S \) using state machine replication.

Figure 6 shows the state machine of replica \( R_i' \in R' \). Upon a do event, \( R_i' \) performs the corresponding transition on \( R_i \)'s state machine and broadcasts any message \( m' \) that \( R_i \) would send to \( S \). Upon receiving a message \( m \), \( R_i' \) delivers \( m \) to the two state machines it maintains. (However, because \( m \) corresponds to a message sent by some \( R_i \), the \( R_i \) state machine ignores it, by Definition 14.) If, as a result of receiving \( m \), \( S \) broadcasts a message \( m^* \), then \( R_i' \) (locally) delivers \( m^* \) to \( R_i \)'s state machine and broadcasts any message \( m' \) that \( R_i \) sends as a result of receiving \( m^* \).

In any execution \( e' \) of \( R' \) that satisfies causal atomic broadcast, all messages are delivered to all replicas in the same order. Therefore, each replica \( R_i' \) performs the same state transitions as \( S \), and (locally) delivers the same messages from \( S \) to its \( R_i \) state machine. The claim follows.

**Client/server lower bound.** Since the executions constructed in the proof of Theorem 5 satisfy causal atomic broadcast, they can also be viewed as executions of a protocol simulating a push-based client/server protocol (Proposition 10). We therefore obtain

**Corollary 11.** Let \( R \) be a push-based client/server protocol that satisfies the weak or strong list specification for \( n \geq 3 \) replicas. Then the worst-case metadata overhead of \( R \) on the clients over executions with \( D \) deletions is \( \Omega(D) \).

**Proof.** Let \( R \) be a push-based client/server protocol that satisfies the weak or strong list specification. Let \( R' \) be the protocol simulating \( R \) from Proposition 10. Take \( D \geq 4 \). By Theorem 5, there exists an execution \( o_D \) of \( R' \) with \( D \) deletions such that: (1) the metadata overhead of some state \( \sigma \) of some replica \( R' \in R' \) is \( \Omega(D) \); (2) \( o_D \) satisfies causal atomic broadcast; and (3) \( R' \) does not receive any message before \( \sigma \). Because \( R' \) simulates \( R \), we have that \( \sigma = ((q^R_i, q^S_i)_j, \ldots) \), where \( q^R \) and \( q^S \) are, respectively, internal states of the replica \( R \in R \) that \( R' \) is simulating and of the server. Moreover, it follows from Proposition 10 that \( q^S \) is the initial internal state of the server. Therefore, \( \|(q^R_i, q^S_i)_j\| = O(|q^R_i| + |q^S_i|) = O(|q^R_i|) \), because \( |q^S| \) is a constant. Since the user-observable content at \( R' \) and \( R \) is the same, it follows that the metadata overhead at \( R \) is \( \Omega(D) \). □

7. RELATED WORK

Previous attempts at specifying the behavior of replicated list objects [16, 27] have been informal and imprecise: they typically required the execution of an operation at a remote replica to preserve the effect of the operation at its original replica, but they have not formally defined the notions of the effect and its preservation.

Burckhardt et al. [4] have previously proposed a framework for specifying replicated data types (on which we base our list specifications) and proved lower bounds on the metadata overhead of several data types. In contrast to us, they handle much simpler data types than a list. Thus, our specifications have to extend theirs with an additional relation, defining the order of elements in the list. Similarly, their proof strategy (and its extension in [2]) for establishing lower bounds would not be applicable to lists; obtaining a lower bound in this case requires a more delicate decoding argument, recovering information incrementally.

There are more protocols implementing a highly available replicated list than the RGA protocol we considered. Treedoc [21] and Logoot [31] are other implementations of the strong list specification using the approach of replicated data types [25]. As in RGA, the state of a replica can be viewed as a tree, where a deterministic traversal defines the order of the list. The replication protocol represents a partial list order, i.e., ordering restrictions inferred at the time user performs operations. The total list order is computed as a view of the graph based on a non-declarative specification of intended ordering.

Another class of protocols is based on operational transformations (OT) [12], which apply certain transformation functions to pairs of concurrent updates. If applying the transformation function allows commuting two operations (TP1) and three operations (TP2) then OT ensures that the list state converges, regardless of the order in which the operations are received [22]. However, it was shown [13] that several OT protocols do not satisfy TP1 and TP2 and do not converge. OT protocols store a log of updates at each replica, so their metadata overhead is also at least linear in the number of updates.

8. CONCLUSION

This paper provides a precise specification of the list replicated object, which models the core functionality of collaborative text editing systems. We define a strong list—and show that it is implemented by an existing system [23]—as well as a weak list, which we conjecture describes the behavior of the Jupiter protocol [19], underlying public collaboration systems [30].

We prove a lower bound of \( \Omega(D) \), where \( D \) is the number of deletions, on the metadata overhead of push-based list protocols, which model the implementation of all highly available list protocols that we are aware of. Our lower bound applies for both weak
and strong semantics. Exploring client/server systems in future research is therefore of practical interest, as some client/server systems do not implement the strong semantics, and our results suggest that this might not offer a complexity advantage.

We also show a simple list protocol whose metadata overhead is \(O(\log k)\), where \(k\) is the number of operations. Closing the gap between the upper and lower bound is left for future work, as is the question of relaxing the restriction to push-based protocols.

Our work is a first step towards specifying and analyzing general collaborative editing systems, providing more features than those captured by the list object. This includes systems for sharing structured documents, such as XML [18]. While our lower bound would hold for more general systems, it is possible that the additional features induce additional complexity.

Acknowledgments

We would like to thank Marc Shapiro and Pascal Urso for comments that helped improve the paper. Attiya and Morrison were supported by the Israel Science Foundation (grant 1749/14) and by Yad Hanadiv Foundation. Gotsman was supported by an EU project ADVENT. Yang was supported by an Institute for Information & Communication Technology Promotion (IITP) grant funded by the Korea government (MSIP, No. R0190-15-2011). Zawirski was supported by an EU project SyncFree.

9. REFERENCES

APPENDIX

A. PROOFS FOR LIST IMPLEMENTATION

Proof of Lemma 3.
This is easy to see for the special case where \( B = A \cup \{(a, t, p)\} \) since \( a \) has no children (because \( A, B \) are TT trees), the depth-first traversal of \( B \) takes the same course as the one for \( A \) except for visiting the node \( a \) and then immediately returning to its parent, thus \( s(B) \) is equal to \( s(A) \) with \( a \) inserted at some position. From this special case we get to the general case by induction, adding one node at a time (there always exists at least one node in \( B \) whose parent is in \( A \) or is \( \alpha \), and can thus be added to \( A \) without breaking the conditions for \( TI \) trees).

Proof of Theorem 1.
First, note that \( R^n_{\alpha} \) does satisfy the basic requirements for a protocol: all replicas accept any operation or message, and deterministically transition. It also satisfies the first condition for a push-based protocol: the send transition is enabled right after an insertion operation.

Let \( \alpha \) be a concrete execution of \( R^n_{\alpha} \) that satisfies causal broadcast. We define a replica order relation \( \alpha \) on the events in \( \alpha \) by letting \( e \rightarrow e' \) denote that \( e \) and \( e' \) are events by the same replica, and \( e \) precedes \( e' \) in \( \alpha \). Also, for two events \( e, e' \in \alpha \), we define the delivery relation to indicate that the first message sent by \( \text{repl}(e) \) after \( e \) is received by \( \text{repl}(e') \) before \( e' \):

\[
(e \rightarrow e') \iff \exists m:\ 
\begin{align*}
& (e \leq_\alpha \text{send}(m)) \land (\text{receive}(m) \leq_\alpha e') \\
& \land \neg(\exists m'. e \leq_\alpha \text{send}(m')) <_\alpha \text{send}(m).
\end{align*}
\]

Without loss of generality, in the following we only consider executions \( \alpha \) of the protocol that deliver all operations to all replicas: for each insertion or deletion operation \( e \), each replica \( R \), there exists an event \( e' \) at \( R \) such that \( e \rightarrow e' \). If \( \alpha \) is infinite, this already follows from our definition of sufficiently connected networks. Otherwise, we can simply append some additional send and/or receive events to \( \alpha \).

To prove correctness and the second condition of push-based protocols, we construct an abstract execution \((H, \text{vis})\) that satisfies all of the following conditions:

- **(O1)** \( \alpha \) complies with \( H \): for all replicas \( R \), \( H|_R = \alpha|_{\text{deliv}} \);
- **(O2)** for \( e, e' \in H \), \( e \rightarrow \text{vis} e' \) if and only if \( e \rightarrow \text{hb} e' \);
- **(O3)** \((H, \text{vis})\) is an element of the strong list specification: \((H, \text{vis}) \in A_{\text{strong}}\).

Construction of \((H, \text{vis})\). Let \( H \) be the subsequence of all do events in \( \alpha \). For the purpose of enabling induction proofs later on, we define the visibility relation \( \text{vis} \) not just on \( H \), but on all events in \( \alpha \). We define visibility as the transitive closure of replica and delivery order: \( \text{vis} = (\alpha \cup \text{deliv})^+ \).

Claim 12. \((H, \text{vis})\) is an abstract execution.

Proof. Following Def. 4, we need to check that \( \text{vis} \) is acyclic. This is obvious because either \( e \rightarrow \text{deliv} e' \) or \( e \rightarrow \text{hb} e' \) implies that \( e <_\alpha e' \), thus \( e <_\alpha \text{vis} e' \) also implies \( e <_\alpha e' \), thus it is acyclic. Condition (1) is satisfied because \( \alpha <_\text{vis} \alpha \). Condition (2) is satisfied because \( e \rightarrow \text{vis} e' \) implies \( e <_\alpha_\text{vis} e' \) and thus \( e <_\alpha_\text{vis} e' \). Condition (3) is satisfied because \( \text{vis} \) is transitive by definition.

Clearly, obligation (O1) is satisfied: \( \alpha \) complies with \( H \) since the order of events in \( H \) matches their order in \( \alpha \). Also, obligation (O2) is satisfied by definition of \( \text{vis} \) and \( \text{hb} \). The nontrivial part is obligation (O3), which requires us to show that \( H \) satisfies the strong list specification (Def. 7).

In the remainder of this section, we prove the conditions of Def. 7 in the order (1c),(1a),(1b),(2). Condition (1c) follows directly from the properties of the data structure. To prove (1a), we need to show that what is stored in \( N \) and \( T \) corresponds to the insertion and deletion operations that are visible. The following two claims do just that, and together imply condition (1a).

For each inserted element \( a \in \text{elems}(H) \), let \( e_a \) be the event of the insert operation, and let \((a, t_a, p_a)\) be the tuple constructed during insertion.

Lemma 13. Let \( e \) be an event in \( \alpha \), and let \((N', \ldots, \ldots)\) be state of the replica \( \text{repl}(e) \) after executing \( e \). Then \( N' \) contains all nodes that were inserted by \( e \) or insertion operations visible to \( e \): \( N' = \{ (a, t_a, p_a) | e_a \leq_\text{vis} e \} \).

Proof. By induction over \( \alpha \) and case distinction on \( e \).

Insert. By the induction hypothesis (or initial state definition, if \( e \) is the first event of the replica), \( N \) in the prestate matches visible insertion operations not counting \( e \) itself. Then \( e \) happens and its tuple is also added to \( N \), thus preserving the invariant.

Receive. \( \text{vis}^{-1}(e) \) contains the union of visible operations \( \text{vis}^{-1}(e') \) of the sending event \( e' \) and the predecessor event on the same replica. Symmetrically, \( e \) is updated to contain the delivered insertion tuples, which capture all insertions between the last send event \( e'' \) of the sender preceding \( e' \) and \( e' \). Because of the causal broadcast guarantee, and by the induction hypothesis, any insertions visible to \( e'' \) must have already been delivered to replica executing \( e \), so the updates correspond.

Others. Neither the visible insertion operations nor \( N \) are updated.

Lemma 14. Let \( e \) be an event of \( \alpha \), and let \((\ldots, T', \ldots)\) be state of the replica \( \text{repl}(e) \) after executing \( e \). Then \( T' \) contains all elements that were deleted by \( e \) or deletion operations visible to \( e \): \( T' = \{ a | \text{do}(\text{del}(a)) \leq_\text{vis} e \} \).

Proof. By induction over \( \alpha \) and case distinction on \( e \).

Delete. By the induction hypothesis (or initial state definition, if \( e \) is the first event of the replica), \( T \) in the prestate matches visible deletion operations not counting \( e \) itself. Then \( e \) happens and its element is also added to \( T \), thus preserving the invariant.

Receive. \( \text{vis}^{-1}(e) \) contains the union of visible deletions in \( \text{vis}^{-1}(e') \) of the sending event \( e' \) and the predecessor event on the same replica. Symmetrically, \( T \) is updated to contain the delivered deletion tuples, which capture all deletions between the last send event \( e'' \) of the sender preceding \( e' \) and \( e' \). Because of the causal broadcast guarantee, and by the induction hypothesis, any insertions visible to \( e'' \) must have already been delivered to replica executing \( e \), so the updates correspond.

Others. Neither the visible deletion operations nor \( T \) are updated.

To prove conditions (1b) and (2) of Def. 7, we need to first define the list order relation. To prepare for this definition, we first observe the following:

Lemma 15. Let \((N, \ldots, \ldots)\) be the state of a replica. Then \( N \) is a TI tree.

Proof. By construction, each inserted node has a unique timestamp (because the timestamp contains the replica identifier and is
larger than all previous timestamps by the same replica). Thus, any of the nodes appearing anywhere in the execution satisfy TI tree conditions (1) and (3). To prove condition (2), note first that \((a, t_a, ...) \xrightarrow{\text{rep}} (b, t_b, ...)\) implies \(e_a \xrightarrow{\text{vis}} b \) by Lemma 13, because the parent is in the set \(N\) of the replica that performs the insertion. Now, consider the tree \(N\) in the post-state of some event \(e\). Using Lemma 13, we see that if \((b, t_b, t_a) \in N\), then \(e_a \xrightarrow{\text{vis}} e \xrightarrow{\text{vis}} e\), thus by transitivity of \(\text{vis}\) also \(e_a \xrightarrow{\text{vis}} e\), thus \((a, t_a, ...) \in N\). □

**List Order.** Now we can define the list order relation: for \(a, b \in \text{elems}(H)\), we let \(a \xrightarrow{\text{by}} b\) if and only if there exists an \(e \in \alpha\) with post-state \((N,\ldots,\ldots)\) such that \(a\) appears before \(b\) in \(s(N)\). It may appear at first that the various \(\text{post}\) sets could lead to contradictory orderings. However, this is not so: this list order satisfies condition (2) of Def. 7:

**Lemma 16.** \(\alpha\) defines a total order on \(\text{elems}(H)\).

**Proof.** Irreflexive. All \(N\) are TI trees by Lemma 15, and thus contain no duplicate elements. Total. Let \(a, b \in \text{elems}(H)\), inserted by events \(e_a, e_b\) respectively. Since \(e_a\) enables a send event, \(\text{rep}(e_a)\) must eventually send a message containing \((a,\ldots,\ldots)\), and \(\text{rep}(e_b)\) must receive it (recall that we only consider executions where every message is delivered), either before or after \(e_a\). After both of those, the replica state on \(\text{rep}(e_b)\) contains in \(N\) nodes for both \(a\) and \(b\), thus \(s(N)\) orders them. Transitive. Let \(a, b, c \in \text{elems}(H)\). Assume that \(s(N_1)\) orders \(a\) before \(b\) and \(s(N_2)\) orders \(b\) before \(c\). Since all insertions are eventually propagated to all replicas, there exists a \(N_3\) in some replica state such that \(\alpha = \{N_1 \subseteq N_3\} \subseteq N_3\). By Lemma 3 and Lemma 15, in \(s(N_3)\), \(a\) appears before \(b\) and \(b\) before \(c\). Thus \(a\) appears before \(c\) in \(s(N_3)\). □

Finally, because returned lists are ordered by \(s(N)\), the list order also satisfies (1b) of Def. 7, which concludes the proof.

**Proof of Theorem 2.** Since timestamps grow with the number of operations performed, and we assume the number \(n\) of replicas is fixed, they can be encoded in \(O(\lg k)\). Also, for simplicity, we assume that elements in \(A\) can be encoded in \(O(\lg k)\) (if the elements in \(A\) are larger than that, we can modify the algorithm to affix an \(O(\lg k)\) identifier to each inserted element, and only keep those identifiers around after elements are deleted).

We consider a replica with state \((N, T, (A, K))\). Let \(w = a_1 \ldots a_n\) be the list represented (i.e. the list returned by a read operation), and let \(s_i\) be the size of \(a_i\). Then, the size of the represented data is \(s_w = \sum_i s_i\). To obtain the metadata overhead, we need to divide the size of the replica state by the size of the represented data. To compute the size of the data in the replica state, observe the following:

- for each element \(a_i\) in the list, \(N\) stores a triple of size \(s_i + O(\lg k) + O(\lg k)\).
- for each element deleted from the list, \(N\) stores a triple of size \(O(\lg k) + O(\lg k) + O(\lg k)\).
- always \(A \subseteq N\) and \(K \subseteq T\) (follows easily from transition rules)

Thus, the respective sizes are

\[
\begin{align*}
    s_N &= \sum_i (s_i + O(\lg k)) = O(\lg k)\sum_i s_i \\
    s_T &= d \cdot O(\lg k) \\
    s_A &\leq s_N \\
    s_K &\leq s_T
\end{align*}
\]

And we get

\[
\frac{s_N + s_T + s_A + s_K}{s_w} \leq \frac{2s_N + 2s_T}{s_w} \leq 2 \frac{O(\lg k)\sum_i s_i + d \cdot O(\lg k)}{(\sum_i s_i)} \leq 2O(\lg k) + d \cdot O(\lg k)) \leq O((\lg k)(1 + d)).
\]

**A.1 Tree/List Equivalence**

In the standard RGA implementation [24], TI trees are represented as lists, and insertion of nodes follows a splicing procedure. On the other hand, our formulation represents TI trees as sets, and insertion of nodes is set union. We now show equivalence of these representation: each TI tree corresponds to an ordered list of pairs (elements, timestamps), in such a way that node insertions into the tree correspond to splicing insertions into the list.

**Tree to List** To obtain a corresponding list from the tree, we use the same traversal as before (depth-first, parents before children, children in order of descending timestamps). Formally, for a TI tree \(N\), and for a parent \(p \in N\), we define a visitor function \(\ell(N)(p)\) that visits the subtree containing \(p\) and all its descendants, and returns a sequence \(\ell(N)(p) \in (U \times C)^{\star}\) of element/timestamp pairs, as follows:

\[
\ell(N)(p) = (p.a, p.t) \ell(N)(c_1) \ldots \ell(N)(c_n)
\]

where \(c_1, \ldots, c_n\) are the children of \(p\) in \(N\), enumerated in the order satisfying \(i < j \Rightarrow c_i.t > c_j.t\). We generalize this notation to allow \(p = \varnothing\), that is, naming the root as a parent, and let \(\varnothing.a = \epsilon\), and \(\varnothing.t = 0\) for some timestamp \(0\) that is smaller than any other timestamp. So, \(\ell(N)(\varnothing)\) is the list representing the whole tree \(N\), and always starts with a dummy element (\(\epsilon, 0\)).

**Splicing** The splicing insertion inserts a tuple \((a', t')\) after a parent \(p\) and after any entries following the parent that have larger timestamp than \(t'\). Formally, we express this using a pattern matching definition

\[
\text{splice}(w, a', t', p') = w_1 (a, p') w_2 (a', t') w_3
\]

where \(w = w_1 (a, p') w_2 w_3\) is a decomposition that satisfies (1) all timestamps appearing in \(w_2\) are larger than \(t'\), and (2) either \(w_3\) is empty, or the first timestamp appearing in \(w_3\) is smaller than \(t'\).

**Representation Equivalence** The following proposition implies that the two representations are equivalent: splicing an element into a list always produces the same result as adding a node to the set.

**Proposition 17.** Let \(N\) and \(N'\) be TI trees such that \(N' = N \cup (a', t', p')\). Then: \(\ell(N)(\varnothing) = \ell(N')(\varnothing)\).

The proposition follows directly from the following inductive formulation.

**Lemma 18.** Let \(N\) and \(N'\) be TI trees such that \(N' = N \cup (a', t', p')\). Then, for all \(p \in N \cup (\varnothing)\) that are ancestors of \((a', t', p')\) in \(N'\), we have \(\text{splice}(\ell(N)(p), a', t', p') = \ell(N')(p)\).

**Proof.** We use induction over the number of descendants of \(p\) in \(N'\), and do a case distinction.

**Case** \(p.t = p'\). Let \(c_1, \ldots, c_n\) be the children of \(p\) in \(N\), enumerated in the order satisfying \(i < j \Rightarrow c_i.t > c_j.t\). Then

\[
\text{splice}(\ell(N)(p), a', t', p') = \text{splice}((p.a, p.t) \ell(N)(c_1) \ldots \ell(N)(c_n), a', t', p') = \text{splice}((p.a, p') w_2 w_3, a', t', p')
\]
where \(w_2 = \ell_N(c_1) \ldots \ell_N(c_i)\) and \(w_3 = \ell_N(c_{i+1}) \ldots \ell_N(c_n)\) with \(i\) chosen such that \(c_i.t > t' > c_{i+1}.t\) (or at least on of those inequalities for the border cases \(i = 0\) or \(i = N\)). Then, (1) either \(w_2\) is empty or all timestamps in \(w_2\) are larger than \(t'\) (because all descendants of the \(c_i\) have larger timestamps than their ancestor, and all the \(c_i\) are larger than \(t'\)), and (2) either \(w_3\) is empty, or the first timestamp in it is \(c_{i+1}.t\), which is smaller than \(t'\). Therefore, this matches the pattern for the \(\text{splice}\) definition, and we get

\[
= (p.a,p') \mathcal{N}(s) \ell_N(c_1) \ldots \ell_N(c_i) \ell_N(c_{i+1}) \ldots \ell_N(c_n)
\]

Now, because in \(N\), the children of \(p\) in descending timestamp order, are \(\{c_1, \ldots, c_i, (a', p', t'), c_{i+1}, \ldots, c_n\}\), we can conclude

\[
= \mathcal{N}(p)
\]

Case \(p \neq p'\). Let \(\{c_1, \ldots, c_n\}\) be the children of \(p\) in \(N\), enumerated in the order satisfying \(i < j \Rightarrow c_i \neq c_j\). Then \(\mathcal{N}(p) = (p.a,p.t) \ell_N(c_1) \ldots \ell_N(c_{i-1}) \ell_N(c_i) \ell_N(c_{i+1}) \ldots \ell_N(c_n)\)

Let \(c_i\) be the child whose subtree contains \((a', p', t')\). There must be exactly one such child because \(N\) is a TI tree, and by the assumption that \((a', p', t')\) is a descendant of \(p\) in \(N\).

\[
= \mathcal{N}(p) \ell_N(c_1) \ldots \ell_N(c_{i-1}) \ell_N(c_i) \ell_N(c_{i+1}) \ldots \ell_N(c_n)
\]

because the subtrees of \(N\) not containing the new element are the same as in \(N\), thus the visit function returns the same result. Applying the induction hypothesis to \(\mathcal{N}(c_i)\), we get

\[
= (p.a,p.t) \ell_N(c_1) \ldots \ell_N(c_{i-1}) \ell_N(c_i) \ell_N(c_{i+1}) \ldots \ell_N(c_n)
\]

and expanding the pattern in \(\text{splice}\) gives us

\[
= (p.a,p.t) \ell_N(c_1) \ldots \ell_N(c_{i-1}) \ell_N(c_i) \ell_N(c_{i+1}) \ldots \ell_N(c_n)
\]

where \(w_1, w_2, w_3\) satisfy \(\ell_N(c_i) = w_1 (a, p') w_2 (a', t') w_3 \ell_N(c_{i+1}) \ldots \ell_N(c_n)\)

- User reads do not change the state of a replica: if \(\Delta(\sigma_1, do(op, v)) = \sigma_2\) and \(op(v) = \text{read}\), then \(\sigma_1 = \sigma_2\).
- \(R\) generates messages only as a result of user operations, and not in response to received messages: \(R\) does not have a message pending in \(\sigma_0\), and if \(\sigma_2 = \Delta(\sigma_1, \text{receive}(m))\) and \(R\) does not have a message pending in \(\sigma_1\), then \(R\) does not have a message pending in \(\sigma_2\).

Write-propagating protocols have an unintended property: whether a message is pending or not can affect the response of high-level operations. We define a class of strictly write-propagating protocols to rule this out.

Definition 16. A protocol \(R\) is strictly write-propagating if it is a write-propagating protocol and for every replica \(R = (Q, M, \Sigma, \sigma_0, E, \Delta)\), if \(\Delta(\sigma_1, \text{send}(m)) = \sigma_2\), then \(\text{do}(op, v)\) is an enabled transition in \(\sigma_1\) if and only if \(\text{do}(op, v)\) is an enabled transition in \(\sigma_2\).

Here, we show that an eventually consistent strictly write-propagating list protocol is necessarily push-based. They are not, however, equivalent. For example, a strictly write-propagating protocol cannot implement a subprotocol to garbage collect metadata entries [23], since this requires sending and responding to messages "spontaneously" and not as a result of a user operation.

Theorem 19. Let \(R\) be an eventually consistent write-propagating protocol satisfying the weak (respectively, strong) list specification. Then \(R\) is a push-based protocol.

Proof. We show that \(R\) satisfies the two properties of a push-based protocol: list insertions and deletions of elements not already deleted cause message generation (Lemma 22) and happens-before being equivalent to visibility in executions satisfying causal broadcast (Lemma 23). Our proof relies on some basic properties, stated in Proposition 20 and Proposition 21 below.

Proposition 20 ([2]). Let \(\alpha\) be a well-formed execution of \(R\) and let \(v\) be an event in \(\alpha\). Then the following sequences of events are well-formed executions of \(R\):

1. \(\beta\), the subsequence of \(\alpha\) consisting of all events \(e'\) such that \(e \Rightarrow e'\).
2. \(\gamma\), the subsequence of \(\alpha\) consisting of all events \(e'\) such that \(e' \Rightarrow e\).

Further, for any replica \(R, \beta|_R\) and \(\gamma|_R\) are prefixes of \(\alpha|_R\).

Proposition 21. Let \(\alpha\) be an execution of \(R\). Let \(A = (H, \text{vis})\) an abstract execution that \(\alpha\) complies with. Let \(o\) be a \(do(op, v)\) event and \(e\) be a \(do(\text{ins}(a,\_\_),\_\_))\) event such that \(a \in v\). Then \(e \Rightarrow o\).

Proof. Suppose the claim is false. We will show that \(R\) does not satisfy the weak (respectively, strong) list specification, which is a contradiction, as follows: Let \(\alpha'\) be the subsequence of \(\alpha\) consisting of all events \(e'\) such that \(e' \Rightarrow e\). By Proposition 20, \(\alpha'\) is a well-formed execution, and \(\alpha'|_R = \alpha|_R\), where \(R = \text{repl}(o)\). However, because inserted elements are unique, there is no insert of \(a\) in \(\alpha'\). The claim follows.

Lemma 22. Let \(\alpha\) be of \(R\) and \(v \in \alpha\). Then (1) if \(\text{op}(v) = \text{ins}(a,\_\_),\_\_), \text{repl}(v)\) has a message pending after \(e\); and (2) if \(\text{op}(v) = \text{del}(a)\) and there does not exist \(e' \Rightarrow e\) with \(\text{op}(e') = \text{del}(a)\), then \(\text{repl}(v)\) has a message pending after \(e\).
\textbf{Proof.} Suppose the claim is false. Let \( \sigma \) be the state of \( R = \text{repl}(e) \) after \( e \) (in which \( R \) does not have a message pending). Let \( \alpha_1 \) be the subsequence of \( \alpha \) consisting of all events \( e' \) such that \( e' \xrightarrow{\text{hb}} e \). By Proposition 20, \( \alpha_1 \) is a well-formed execution, and \( \alpha_1[R] \) if a prefix of \( \alpha[R] \). Thus, \( R' \) is not satisfied at the end of \( \alpha_1 \); let \( \beta \). Let \( \beta = \alpha_1 \times \alpha_2 \) be an execution of \( \mathcal{R} \) obtained by appending receive events to \( \alpha_1 \) in some arbitrary order, so that \( \beta \) every replica receives every message sent by another replica in \( \alpha_1 \). Because \( \mathcal{R} \) has op-driven messages, it follows that \( \beta \) does not have a message pending at the end of \( \beta \). Moreover, no replica \( R' \neq R \) has a message pending at \( \beta \). If there are no events at \( \beta' \), then it does not have a message pending at the end of \( \alpha_1 \). Otherwise, consider the last event \( e' \) at \( \alpha_ \). Then \( e' \xrightarrow{\text{hb}} e \), by definition of \( \alpha_1 \), and so \( \beta' \) sent a message at or after \( \beta' \). But \( e' \) is the last event at \( \alpha' \), so \( e' \) is a send event.

Consider the infinite execution of \( \mathcal{R}, \beta_\infty = \beta_1 \beta_2 \ldots \), where the \( \beta_i \) are reads at some \( R' \neq R \). Let \( \Lambda = (H, \text{vis}) \) be an abstract execution satisfying eventual visibility \( \beta_\infty \) complies with. Then there exists some \( r_j \) such that \( e \xrightarrow{\text{vis}} r_j \). We now consider the two possible cases:

(1) \( \text{op}(e) = \text{ins}(a, \_\_ \_ ) \). Because \( e \xrightarrow{\text{hb}} r_j \), by Proposition 21, \( a \notin \text{rval}(r_j) \). It follows from the weak list specification that there exists a \( \text{del}(a) \) event, \( d \in H \), such that \( d \xrightarrow{\text{vis}} r_j \). It follows from our assumption that users delete only elements that appear in the response of a preceding operation on the same replica that \( a \in \text{rval}(e') \) for some \( e' \xrightarrow{\text{hb}} d \). By Proposition 21, \( e \xrightarrow{\text{hb}} e' \). But this is a contradiction, since \( e' \xrightarrow{\text{hb}} e \).

(2) \( \text{op}(e) = \text{del}(a) \). It follows from our assumption that users delete only elements that appear in the response of a preceding operation on the same replica that \( a \in \text{rval}(f) \) for some \( f \xrightarrow{\text{vis}} e \). Therefore, there exists an event \( d = \text{inv}(a, \_\_ \_ ) \in H \), such that \( d \xrightarrow{\text{vis}} e \), and so \( d \xrightarrow{\text{vis}} e \xrightarrow{\text{vis}} r_j \). It follows from the weak list specification that \( a \notin \text{rval}(r_i) \) for every \( i \geq j \). Observe, however, that the execution \( \beta_\infty \) obtained by removing \( e \) from \( \beta_\infty \) is a well-formed execution of \( \mathcal{R} \), because the only events at \( R \) after \( e \) are message receipts. Now, let \( \Lambda' = (H', \text{vis}') \) be an abstract execution satisfying eventual visibility \( \beta_\infty \) complies with. Then there exists some \( r_i \), \( i \geq j \), such that \( d \xrightarrow{\text{vis}} r_i \). Since \( a \notin \text{rval}(r_i) \), there exists a \( \text{del}(a) \) event, \( e' \in H \), such that \( d \xrightarrow{\text{vis}} e' \xrightarrow{\text{vis}} \text{rval}(r_i) \). Thus, \( e' \in \beta_\infty \) and so \( e' \xrightarrow{\text{hb}} e \), a contradiction. \( \square \)

\textbf{Lemma 23.} Let \( \alpha \) be an execution of \( \mathcal{R} \) that satisfies causal broadcast. Then there exists an abstract execution \( \Lambda = (H, \text{vis}) \in \mathcal{A}_{\text{weak}} \) (respectively, \( \Lambda \in \mathcal{A}_{\text{strong}} \)) that \( \alpha \) complies with, such that for all \( e' \), \( e \in H \), \( e' \xrightarrow{\text{vis}} e \) if and only if \( e' \xrightarrow{\text{hb}} e \).

\textbf{Proof.} It is immediate that there exists an abstract execution \( \Lambda = (H, \text{vis}) \) that \( \alpha \) complies with such that for all \( e' \), \( e \in H \), \( e' \xrightarrow{\text{vis}} e \) if and only if \( e' \xrightarrow{\text{hb}} e \). To show that \( \alpha \in \mathcal{A}_{\text{weak}} \) (respectively, \( \alpha \in \mathcal{A}_{\text{strong}} \)), we must show is satisfies all conditions in Definition 8 (respectively, Definition 7). Conditions 1b and 1c in Definition 7, as well as Condition 2 in Definition 8 (respectively, Definition 7) are satisfied because, by assumption, \( \mathcal{R} \) satisfies the weak (respectively, strong) list specification, and these conditions depend only on the operations and their responses, and not on the visibility relation.

It remains to show that for every \text{do} event \( e \in \alpha \), \( \text{rval}(e) = L(e, \text{vis}) \), where

\[ L(e, \text{vis}) = \{ a \mid (\text{do}(\text{ins}(a, \_\_ \_ )) \xrightarrow{\text{vis}} e) \land \neg(\text{do}(\text{del}(a, \_\_ \_ )) \xrightarrow{\text{vis}} e) \} \]