Specification and Complexity of Collaborative Text Editing

Hagit Attiya
Technion

Sebastian Burckhardt
Microsoft Research

Alexey Gotsman
IMDEA Software Institute

Adam Morrison
Technion

Hongseok Yang
University of Oxford

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 [7, 8], Apache Wave [1], or wikis [16], 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 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 [29, 30].

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 $a$ at the left of $x$, and to $R_3$, which inserts $b$ at 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 $axb$ and $a^x b$ established at other replicas. However, some implementations allow $ba$ as a response; e.g., this is the case in a Jupiter protocol [20], used in public collaboration systems [31].

There have been a number of proposals of highly available collaborative editing protocols, using techniques such as operation transformations [13, 23, 27, 28] and replicated data types (aka CRDTs) [22, 24, 32]. However, specifications of their desired behavior [17, 28] 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 [14]. 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 [13] (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

*Now at Google.

Copyright © Owner/Authors, 2016. This is the authors’ version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in PODC’16, July 25–28, 2016, Chicago, IL, USA, http://dx.doi.org/10.1145/2933057.2933090.

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

© 2016 Copyright held by the owner/authors(s). Publication rights licensed to ACM.

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 [24], which is in the style of replicated data types [22] (Section 4). The protocol represents the list as a tree, with read operations traversing the tree in a deterministic order. Inserting an element \( a \) right after an element \( x \) (as in Figure 1(b)) adds \( a \) as a child of \( x \) 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 [5] 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 \( O(D \lg k) \), 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 [32] 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.\(^1\)

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 incurs 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 [10]. 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 \( \sigma_w \) 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 \( \sigma_w \); the messages sent in the reconstructed part of \( \sigma_w \) 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 [3, 5], which supports a set of operations \( \text{Op} \). 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:

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

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

We require that a \( \text{send}(m) \) event is enabled in state \( \sigma \) if and only if \( \sigma = (q, m) \) for \( m \neq \bot \), and in this case \( \Delta((q, m)) = (q', \bot) \) for some \( q' \). We further require that a replica can execute any operation with its return values computed deterministically: for any operation \( op \in \text{Op} \), exactly one \( \text{do}(op, v) \) event is enabled in \( \sigma \). We also require that a replica can accept any message: for any message \( m \), \( \text{receive}(m) \) 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;\(^2\) 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 \( \mathcal{R} \).\(^3\) For each event \( e \), we let \( \text{repl}(e) \in \mathcal{R} \) be the replica at which it occurs, and for each do event \( e = \text{do}(op, v) \) we let \( \text{op}(e) = op \) and \( \text{val}(e) = v \). 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 \( \text{receive}(m) \) event at \( R \) is preceded by a \( \text{send}(m) \) event in \( \alpha \).

Let \( \alpha \) be an execution. Event \( e \in \alpha \) happens before event \( e' \in \alpha \) [15] (written \( e \xrightarrow{\text{hb}(\alpha)} e' \), or simply \( e \xrightarrow{\text{hb}} e' \) if the context is clear) if one of the following conditions holds: (1) Thread of execution: \( \text{repl}(e) = \text{repl}(e') \) and \( e \) precedes \( e' \) in \( \alpha \). (2) Message delivery: \( e = \text{send}(m) \) and \( e' = \text{receive}(m) \). (3) Transitivity: There is an event \( f \in \alpha \) such that \( e \xrightarrow{\text{hb}} f \) and \( f \xrightarrow{\text{hb}} e' \).

\(^1\)Wikipedia stores this information also to track the document’s edit history.

\(^2\)The latter is used to support atomic broadcast [10], defined later.

\(^3\)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 \( \alpha'\parallel R \), 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 \( \mathcal{R} \) 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 [24,27]. We model this by considering only executions that satisfy causal broadcast [6]:

**Definition 2.** An execution \( \alpha \) of a protocol \( \mathcal{R} \) satisfies causal broadcast if for any messages \( m, m' \), whenever send\((m)\) \( \not\in \mathcal{R} \), 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 [10] 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 [6]; 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 [13], 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.
- \( \text{read} \): 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. [5] for specifying replicated data types [26]. We specify the list object by a set of abstract executions, which record the operations performed by users (represented by \( \text{do} \) events) and visibility relationships between them. Since collaborative editing systems generally preserve causality between operations [27], 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\(^4\), 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 \( \mathcal{A}' = (H', \text{vis}') \) is a prefix of abstract execution \( A \) if: (1) \( H' \) is a prefix of \( H \); and (2) \( \text{vis}' = \text{vis} \cap (H' \times H' \cup H' \times H) \). A specification of an object is a prefix-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 \( A = (H, \text{vis}) \) if for every replica \( R \in \mathcal{R} \), \( H'_{\mid R} = \alpha'_{\mid R}^R \), where \( \alpha'_{\mid R}^R \) denotes the subsequence of do events by replica \( R \) in \( \alpha \).

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

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 \( A = (H, \text{vis}) \):

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

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 \( e_1 = e_2 \vee e_1 \xrightarrow{\text{vis}} e_2 \).

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

\[\]
1. Each event \( e = do(op, w) \in H \) returns a sequence of elements \( w = a_0 \ldots a_n\), where \( a_i \in \text{elems}(A) \), such that

(a) \( w \) contains exactly the elements visible to \( e \) that have been inserted, but not deleted:

\[
\forall a, a \in w \iff (\text{do}(\text{ins}(a, \_), \_)) \leq_{\text{vis}} e \land -(\text{do}(\text{del}(a, \_)) \leq_{\text{vis}} e).
\]

(b) The order of the elements is consistent with the list order:

\[
\forall i, j. (i < j) \implies (a_i, a_j) \in \text{lo}.
\]

(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 [24]. However, some protocols, such as Jupiter [20], 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)\(^5\).

**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 \( \text{lo} \subseteq \text{elems}(A) \times \text{elems}(A) \) such that:

1. Condition 1 in Definition 7 is satisfied.

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

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 [29, 30]. 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 \( -(e' \rightarrow e) \).

\(^5\)We 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 \( a \) 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 [5] of a state \( \sigma = (q, m) \) is \( |q| / |w| \) for the unique \( w \) such that \( \text{do}(\text{read}, w) \) is enabled in \( \sigma \); here \( w \) represents the user-observable contents of \( \sigma \). Note that the contents of the send buffer is not part of the metadata.

**Definition 11 ([5]).** 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 [24], 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 \( a \in U \) is the element, \( t \in C \) is a timestamp for the insertion, and \( p \in (C \cup \{\_\}) \) is either the parent node (identified by its timestamp) or the symbol \( \_ \) 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 \( n.p = p \). Two nodes \( n, n' \) with \( n'.p = n.t \), we say \( n \) is the parent of \( n' \) and write \( n \rightarrow n' \).

**Definition 12.** A set of nodes \( N \) is a TI tree if (1) timestamps uniquely identify nodes: \( \forall n, n' \in N : n.t = n'.t \implies n = n' \); (2) all parents are present: if \( n \in N \) and \( n.p \neq \_ \), then \( n'.p \rightarrow n \) for some \( n' \in N \); and (3) parents are older than their children: \( n \rightarrow n' \implies n.t < n'.t \).

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.
4.2 The RGA Protocol

We now define the RGA protocol $R_{\text{rga}}^n$ 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:

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

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 $\perp$ 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 $((1 + (the\-largest\ timestamp\ in\ N)), i)$, 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 nonempty. 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 $A_m$.

Theorem 1. The protocol $R_{\text{rga}}^n$ satisfies the strong list specification.

Theorem 2. The worst-case metadata overhead of $R_{\text{rga}}^n$ 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 [2, §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 [25], TI trees are represented as lists (corresponding to the tree traversal). We show in [2, §A] 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 [20, 22, 24, 27], 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 causality. 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 \( e = do(ins(a, \_), \_) \in \alpha \), replica repl(\( e \)) has a message pending after \( e \).
- For any execution \( \alpha \) of \( \mathcal{R} \) and \( e = do(del(a, \_), \_) \in \alpha \), if there does not exist event \( e' = do(del(a, \_), \_) \in \alpha \) 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 \( A = (H, vis) \in \mathcal{A}_{weak} (A \in \mathcal{A}_{strong}) \) that \( \alpha \) complies with, such that \( \forall e', e \in H, e' \xrightarrow{vis} e \iff e' \xrightarrow{\_} e \).

The class of push-based protocols contains both op-based protocols [5], in which a message carries a list of the latest operations that the sender has performed (e.g., RGA), and state-based [5] protocols, in which a message describes all operations the sender knows about (i.e., its state). We also show ([2, §B]) that the class of push-based protocols contains eventually consistent write-propagating protocols [3]—which model many deployed highly available eventually consistent protocols [4, 5, 9, 11, 12, 18, 26, 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 of \( \mathcal{R} \) 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 \( 2^d \) of them, 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 \( e_i \) correspond to a state transition \( e \) at replica \( R_i \). We abuse notation by writing op instead of \( do(op, \_\_) \), 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 \( m_i \). Then we insert the string \([1]_i \), immediately to the left or to the right of \([i - 1]_{i - 1} \), 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 \([a]_i \) and \([a]_i \), 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 \([0]_0 [2]_2 [1]_1 \). We then delete all the \([a]_i \) and \([a]_i \) 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 \( A = (H, vis) \) that the encoding execution \( \alpha_w \) complies with, all ins 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 Recover() that recovers \( w_i \) from \( \sigma_w \) and \( m_1, \ldots, m_i \). (We describe 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 replicas 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 $v_{w_i}$; we will show that $[i-1], i \in v_{w_i}$. Next, $R_2$ inserts element $x$ between $[i-1]$ and $i$ and broadcasts a message $m_x$, which we deliver to both replicas. Finally, we read at $R_1$ and observe the list in state $y_{w_i}$. We will show that $y_{w_i}$ contains only $x$ and $*$, and if $x$ precedes $*$ 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 $\alpha_w$, 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 $\alpha_w$ of the form:

$$\beta_w = \alpha_w \quad \text{receive}_2(m_1) \text{ receive}_1(m_1) \ldots \text{ receive}_2(m_i) \text{ receive}_1(m_i) \quad \text{do}_2(\text{read}_2, v_{w_i}) \quad \text{do}_2(\text{ins}_2(x, k_x^0), \ldots) \quad \text{send}_2(m_x) \quad \text{receive}_2(m_x) \quad \text{receive}_1(m_x) \quad \text{do}_1(\text{read}_1, y_{w_i}),$$

where $k_x^0$ 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 $[i-1], i_1 \in v_{w_i}$ (Claim 8), and that $y_{w_i} = x*$ or $y_{w_i} = *x$ (Claim 9). To this end, we exploit the fact that $\sigma_w$ is $R_1$’s state at the end of $\alpha_w$, which allows Recover() to perform the same state transitions at $R_1$ that occur in $\beta_w$, without having access to the entire execution $\alpha_w$ 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 $\alpha_w$. The claim thus follows from the following: (1) $\alpha_w$ is well-formed; (2) at the end of $\alpha_w$, $R_1$ is in state $\sigma_w$; (3) because $R_2$ does not participate in $\alpha_w$, the state of $R_2$ at the end of $\alpha_w$ 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.** $[i-1], i \in v_{w_i}$.

**Proof.** For $j = 1, \ldots, d + 1$, let $e_j, f_j \in \alpha_w$ be the $do$ events in which $R_1$ inserts $[i-1]$ and $[j]$, into the list. Let $r \in \beta_w$ be the $do$ event at which $R_2$ reads $v_{w_i}$. 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 $e_j \xrightarrow{do} r$ and $f_j \xrightarrow{do} r$ 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 $v_{w_i} = \ldots [i-1], i, \ldots$.

**Claim 9.** $y_{w_i} = x*$ or $y_{w_i} = *x$.

**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 $y_{w_i}$. Because
Figure 5: Examples of the encoding and decoding procedures from Theorem 5 applied to $w = 10$. Figure 5(a) shows the execution $\alpha_w$ constructed by the encoding procedure, whose output is $\sigma_w$, the state of $R_1$ at the end of $\alpha_w$. Figure 5(b) shows the first step of decoding $w$ from $\sigma_w$. The decoding procedure performs the state transitions of the events at $R_1$ that are outside of the dashed rectangle and of the events at $R_2$; these transitions are valid because they occur in the depicted execution, in which the events inside the dashed rectangle lead $R_1$ to state $\sigma_w$. The relative order of $x$ and $* \text{ read at } R_1$ therefore recovers bit $w_1 = 1$. Figure 5(c) shows the second step of decoding $w$: recovering $w_1 = 1$ allows the decoding procedure to perform the state transitions by $R_1$ in $\alpha_w$ that depend on $w_1$.

$\mathcal{R}$ is a correct push-based protocol, $\beta'_w$ compiles with some abstract execution $A = (H, \text{vis}) \in \mathcal{A}_{\text{weak}}$ such that $f \xrightarrow{\text{vis}} r$. Now, let $e \in \beta'_w$ be the do event at which $R_1$ inserts $*$ into the list. Then $e \xrightarrow{\text{vis}} r$ by definition of an abstract execution (Definition 4). Because all other elements inserted in $\beta'_w$ are deleted by $R_1$ before $r$, but $x$ and $*$ are not deleted in $\beta'_w$, the claim follows.

Correctness of recovering $w_i$. Having shown that the state transitions performed by Recover() yield the lists $y'_w = x \star$ or $y'_w = \star x$, it remains to show that we correctly recover $w_i$ from $y'_w$: $(y'_w = x \star) \iff (w_i = 1)$.

In principle, the weak list specification allows $R_1$’s read to order $x$ and $\star$ 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 $[i_{-1}]x[\ldots]_{i_{-1}}\text{ and } *_{i_{-1}}$, and together, 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):

$$\gamma'_w = \beta'_w \text{ receives}_3(m_1) \ldots \text{ receive}_3(m_{d_3}) \text{ do}_3(\text{read}_3, z^i_w).$$

We show that the list $z^i_w$ contains $*$ after $x$ if and only if $w_i = 1$. Informally, this follows because every element inserted from iteration $i$ onwards in the encoding procedure (and hence in $\gamma'_w$), including $*$, 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 $*$.

Formally, consider the following events in $\gamma'_w$: $x \star$, the ins of $*$ by $R_1$; $w_x$, the ins of $x$ by $R_2$; and $r_x$, the read by $R_3$, whose response is $z^i_w$. Because $\mathcal{R}$ 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 $e' \xrightarrow{\text{hb}} e$. Therefore, no del event is visible to $w_x, w_*$ or $r_x$. Let $lo$ be a list order that $A$ is consistent with (Definition 6). We proceed to show that $(x, *) \in lo$ 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 $(x, *) \in lo$ if and only if $w_i = 1$. The response of $w_i$ establishes that $(x, *) \in lo$ if and only if $w_i = 1$, since $[i_{-1}], x, [\ldots]_{i_{-1}}, \star \in z^i_w$ and $lo$ is total and transitive on $\{a \mid a \in \text{val}(r_x) = z^i_w\}$.

We conclude by noting that $y'_w = x \star$ or $y'_w = \star x$ (Claim 9); recall that $y'_w$ is the response to the read at $R_3$ performed by Recover(). Since $(x, *) \in lo$ if and only if $w_i = 1$, then $y'_w = x \star$ 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 [20].) 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 $R_i = (Q^i, M, \Sigma^i, \sigma^i_0, \Delta^i)$, 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 [15].

**Proposition 10.** Let $\mathcal{R} = \{R_1, \ldots, R_n, S\}$ be a client/server protocol. Then there exists a protocol $\mathcal{R}' = \{R'_1, \ldots, R'_n\}$
<table>
<thead>
<tr>
<th>State</th>
<th>Event $e$</th>
<th>New state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$((r, s), m)$</td>
<td>do(op, v)</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>send(m)</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>receive(m)</td>
<td>$</td>
</tr>
</tbody>
</table>

that simulates $\mathcal{R}$ in the following sense: (1) for any execution $\alpha'$ of $\mathcal{R}'$ that satisfies causal atomic broadcast, there exists an execution $\alpha$ of $\mathcal{R}$ such that $\alpha|_{R_i} = \alpha'|_{R_i}$ for $i = 1..n$; (2) the set of internal states of each $R'_i$ is $Q^i \times Q^S$, where $Q^i$ and $Q^S$ are respective sets of $R_i$ and $S$; and (3) until $R_i'$ receives a message, its state is $\{\bot, \bot\}$, where $\{q_0^S, \bot\}$ is the initial internal state of $S$.

Proof. For $1 = 1..n$, replica $R'_i \in \mathcal{R}'$ 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 $\mathcal{R}'$ in the same order to simulate the server $S$ using state machine replication.

Figure 6 shows the state machine of replica $R'_i \in \mathcal{R}'$. 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 the message 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 $\alpha'$ of $\mathcal{R}'$ 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 at $S$, 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 $\mathcal{R}$ 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 $\mathcal{R}$ on the clients over executions with $D$ deletions is $\Omega(D)$.

Proof. Let $\mathcal{R}$ be a push-based client/server protocol that satisfies the weak or strong list specification. Let $\mathcal{R}'$ be the protocol simulating $\mathcal{R}$ from Proposition 10. Take $D \geq 4$. By Theorem 5, there exists an execution $\sigma_D$ of $\mathcal{R}'$ with $D$ deletions such that: (1) the metadata overhead of some state $\sigma$ of some replica $R' \in \mathcal{R}'$ is $\Omega(D)$; (2) $\sigma_D$ satisfies causal atomic broadcast; and (3) $R'$ does not receive any message before $\sigma$. Because $\mathcal{R}'$ simulates $\mathcal{R}$, we have that $\sigma = ((q^R_i, q^S_i)_\sigma)$, where $q^R$ and $q^S$ are, respectively, internal states of the replica $R \in \mathcal{R}$ 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, $\|q^R_i, q^S_i\| = O(\|q^R_i\| + \|q^S_i\|) = O(\|q^R_i\|)$, 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 [17, 28] 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. [5] 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 [3]) 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 [22] and Logoot [32] are other implementations of the strong list specification using the approach of replicated data types [26]. 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 position of a node in the tree as a sequence of edge labels on the path from the root of the tree (in RGA, it is a relative position to an existing node). Like RGA, these protocols have worst-case metadata overhead linear in the number of deletions. WOOT [21] is a graph-based list implementation: its main component is a representation of 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) [13], 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 [23]. However, it was shown [14] 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 [24]—as well as a weak list, which we conjecture describes the behavior of the Jupiter protocol [20], underlying public collaboration systems [31].

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 $O(D \log k)$, 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. 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 Institute for Information & Physical Clocks. In SoCC, 2004.

9. REFERENCES