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.

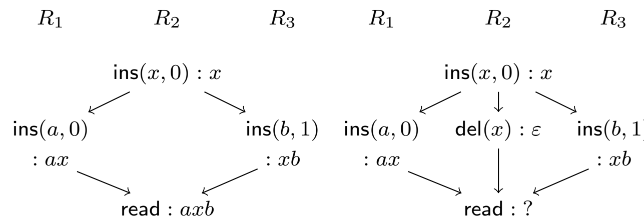
A major factor determining the efficiency and practical feasibility of a collaborative text editing protocol is the space overhead of the metadata that the protocol must maintain to ensure correctness. We show that for a large class of list protocols, implementing *either the strong or the weak list 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 [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].



(a) (b)

Figure 1: Example scenarios of collaborative text editing. Events are presented in format "operation : return value". An arrow from an event e' to an event e expresses that the effects of e' get incorporated at e's replica before e executes.

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_2 to return ab, given the orderings ax and xb 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

*Now at Google.

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

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 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* [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 $O(D \lg k)$, 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 [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.[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 [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 α_w of the protocol such that, at its end, the user-observable state σ_w of some replica is a list of size $O(1)$ bits. We then show that, given σ_w, we can decode w by exercising the protocol in a black-box manner. This implies that all states σ_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 σ_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 α_w; the messages sent in the reconstructed part of α_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 $\Delta : \Sigma \times E \rightharpoonup \Sigma$ 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 σ. Transitions determined by Δ 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 $op \in \mathsf{Op}$ 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 σ 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 $op \in \mathsf{Op}$, exactly one $do(op, v)$ event is enabled in σ. We also require that a replica can accept any message: for any message m, $receive(m)$ is enabled in σ. 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 $\mathsf{repl}(e) \in \mathcal{R}$ be the replica at which it occurs, and for each *do* event $e = do(op, v)$ we let $\mathsf{op}(e) = op$ and $\mathsf{rval}(e) = v$. 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 σ_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 α.

Let α be an execution. Event $e \in \alpha$ *happens before* event $e' \in \alpha$ [14] (written $e \xrightarrow{\mathsf{hb}(\alpha)} e'$, or simply $e \xrightarrow{\mathsf{hb}} e'$ if the context is clear) if one of the following conditions holds: *(1) Thread of execution:* $\mathsf{repl}(e) = \mathsf{repl}(e')$ and e precedes e' in α. *(2) Message delivery:* $e = send(m)$ and $e' = receive(m)$. *(3) Transitivity:* There is an event $f \in \alpha$ such that $e \xrightarrow{\mathsf{hb}} f$ and $f \xrightarrow{\mathsf{hb}} e'$.

[1] Wikipedia stores this information also to track the document's edit history.

[2] The latter is used to support atomic broadcast [9], 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* α if R's has a $send(m)$ event enabled in the state at the end of $\alpha'|_R$, where α' is the prefix of α ending with e.

DEFINITION 1. *The network is* sufficiently connected *in an infinite well-formed execution* α *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* α*, then* R *also sends a message infinitely often in* α*, 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* α *of a protocol* $\mathcal{R}$ *satisfies* causal broadcast *if for any messages* m, m'*, whenever* $send(m) \xrightarrow{\mathsf{hb}} send(m')$*, 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* α *of protocol* $\mathcal{R}$ *satisfies* causal atomic broadcast *if the following conditions hold:* (1) Causal broadcast: α *satisfies causal broadcast.* (2) No duplicate delivery: *each* $send(m)$ *event in* α *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:

- $\mathsf{ins}(a, k)$ 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.
- $\mathsf{del}(a)$ 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, vis)*, where* H *is a sequence of do events*[4]*, and* $\mathsf{vis} \subseteq H \times H$ *is an acyclic* visibility relation *(with* $(e_1, e_2) \in \mathsf{vis}$ *denoted by* $e_1 \xrightarrow{\mathsf{vis}} e_2$*) such that: (1) if* e_1 *precedes* e_2 *in* H *and* $\mathsf{repl}(e_1) = \mathsf{repl}(e_2)$*, then* $e_1 \xrightarrow{\mathsf{vis}} e_2$*; (2) if* $e_1 \xrightarrow{\mathsf{vis}} e_2$*, then* e_1 *precedes* e_2 *in* H*; and (3)* vis *is transitive (if* $e_1 \xrightarrow{\mathsf{vis}} e_2$ *and* $e_2 \xrightarrow{\mathsf{vis}} e_3$*, then* $e_1 \xrightarrow{\mathsf{vis}} e_3$*).*

Figure 1 graphically depicts abstract executions, where vis is the transitive closure of arrows in the figure and H is the result of some topological sort of vis. An abstract execution $A' = (H', \mathsf{vis}')$ is a *prefix* of abstract execution A if: (1) H' is a prefix of H; and (2) $\mathsf{vis}' = \mathsf{vis} \cap (H' \times H')$. 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* α *of a protocol* $\mathcal{R}$ complies with *an abstract execution* $A = (H, \mathsf{vis})$ *if for every replica* $R \in \mathcal{R}$*,* $H|_R = \alpha|_R^{do}$*, where* $\alpha|_R^{do}$ *denotes the subsequence of do events by replica* R *in* α*.*

DEFINITION 6. *A protocol* $\mathcal{R}$ satisfies *a specification* $\mathcal{S}$ *if every execution* α *of* $\mathcal{R}$ *complies with some abstract execution* $A \in \mathcal{S}$*.*

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

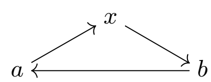
$$\mathsf{elems}(A) = \{a \mid do(\mathsf{ins}(a, \_), \_) \in H\}.$$

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 $e_1 \leq_{\mathsf{vis}} e_2$ for $e_1 = e_2 \vee e_1 \xrightarrow{\mathsf{vis}} e_2$.

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

[4]Formally, H consists of do events instrumented with unique event ids and replicas, as in the case of an execution α. To avoid clutter, we do not use this more accurate presentation.

1. *Each event $e = do(op, w) \in H$ returns a sequence of elements $w = a_0 \ldots a_{n-1}$, where $a_i \in \mathsf{elems}(A)$, such that*
   (a) *w contains exactly the elements visible to e that have been inserted, but not deleted:*
   $$\forall a.\, a \in w \iff (do(\mathsf{ins}(a, \_), \_) \leq_{\mathsf{vis}} e) \wedge \neg(do(\mathsf{del}(a), \_) \leq_{\mathsf{vis}} e).$$
   (b) *The order of the elements is consistent with the list order: $\forall i, j.\ (i < j) \implies (a_i, a_j) \in \mathsf{lo}$.*
   (c) *Elements are inserted at the specified position: if $op = \mathsf{ins}(a, k)$, then $a = a_{\min\{k, n-1\}}$.*
2. *The list order* 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 $a \to x \to b$. 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)[5].

DEFINITION 8. *An abstract execution $A = (H, \mathsf{vis})$ belongs to the* weak list specification $\mathcal{A}_{\mathsf{weak}}$ *if and only if there exists a relation $\mathsf{lo} \subseteq \mathsf{elems}(A) \times \mathsf{elems}(A)$ such that:*

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

Unlike the strong specification, the weak one allows the list order 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: $a \to x$ and $x \to b$.

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 $\mathcal{A}_{\mathsf{strong}}$ or $\mathcal{A}_{\mathsf{weak}}$, 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 $A = (H, \mathsf{vis})$ satisfies* eventual visibility *if for every event $e \in H$, there are only finitely many events $e' \in H$ such that $\neg(e \xrightarrow{\mathsf{vis}} e')$.*

[5] We conjecture that Jupiter satisfies the weak specification.

DEFINITION 10. *A protocol $\mathcal{R}$ satisfying the weak (resp., strong) list specification* guarantees eventual visibility *if every execution α of $\mathcal{R}$ complies with some abstract execution $A \in \mathcal{A}_{\mathsf{weak}}$ (respectively, $A \in \mathcal{A}_{\mathsf{strong}}$) 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 $\sigma = (q, m)$ is $|q|/|w|$ for the unique w such that $do(\mathsf{read}, w)$ 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 $a \in U$ is the element, $t \in C$ is a *timestamp* for the insertion, and $p \in (C \cup \{\circ\})$ 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 $n \xrightarrow{\mathsf{pa}} n'$.

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 $n.p \neq \circ$, then $n' \xrightarrow{\mathsf{pa}} n$ for some $n' \in N$; and (3) parents are older than their children: $n \xrightarrow{\mathsf{pa}} n' \implies n.t < n'.t$.*

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.

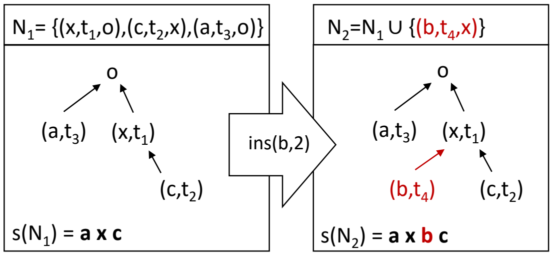


Figure 2: Illustration of TI trees. Each box shows a tree, with the set of nodes that define it, its graphical representation, and the sequence of elements it denotes. The tree on the right results from an insert operation for element b at position 2. The order on the timestamps is $t_1 < t_2 < t_3 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

Insert. To insert a new element a at position k into the list, let $s(N) = a_0 \dots a_{n-1}$ 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 $p = \circ$ (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 $\mathcal{R}^n_{\mathrm{rga}}$ 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 $i \in \{1, \dots, n\}$ is a replica identifier. They are ordered lexicographically:

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

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 $((1 + (\text{the largest timestamp in } N)), i)$, 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 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.

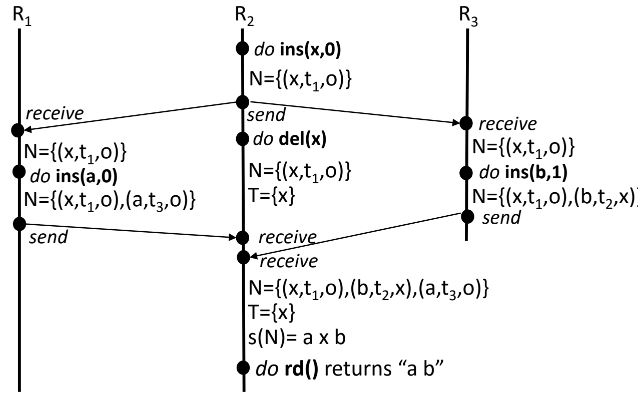


Figure 3: Illustration of an RGA execution, with time proceeding from top to bottom. Bullets show the transitions of the replicas R_1, R_2, and R_3, and in some places we indicate the current state of the tree N and the tombstone set T. The order on the timestamps is $t_1 < t_2 < t_3$.

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 $\mathcal{R}^n_{\mathrm{rga}}$ satisfies the strong list specification.*

THEOREM 2. *The worst-case metadata overhead of $\mathcal{R}^n_{\mathrm{rga}}$ over executions with k operations and D deletions is $O(D \lg k)$.*

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

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* [4] 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 α_D of $\mathcal{R}$ with D deletions such that: (1) the metadata overhead of some state σ of some replica R in α_D is $\Omega(D)$; (2) α_D satisfies causal atomic broadcast; and (3) R does not receive any message before σ in α_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 α_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 α_w, the user-observable list at R_1 contains the single element "$*$" and R_1 has no messages pending; and yet (3) we can decode w given only σ_w, the state of R_1 at the end of α_w (this decoding process exercises the protocol $\mathcal{R}$ in a black-box manner). Hence, all states σ_w must be distinct. Since there are 2^d of them, one of these states σ_{w_0} must take at least d bits. Since this state has no messages pending, its metadata overhead is $\Omega(D)$, and thus, α_{w_0} is the desired execution.

Encoding w. Given a d-bit string $w = w_1 \dots w_d$, we construct an execution α_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 α_w for $w = 10$.

Only replica R_1 participates in the encoding execution α_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 $i = 1, \dots, d$. Each step i begins with R_1 in state σ_i having a message m_i pending. R_1 first broadcasts m_i. We then insert the string $[_i\,]_i$ 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 $[_d$ 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 $[_0\,]_0\,[_2 *]_2\,[_1\,]_1$. We then delete all the $[_i$ and $]_i$ elements, for $i = 0, \dots, d$, and if D is odd, we insert and delete an additional element, so that the number of deletions in α_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 $A = (H, \mathsf{vis})$ that the encoding execution α_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 σ_w, the state of R_1 at the end of the encoding execution α_w. It is easy to check that α_w is well-formed; furthermore, it vacuously satisfies causal atomic broadcast.

Decoding w from σ_w. We reconstruct w one bit at a time by "replaying" the execution α_w. To replay iteration i of α_w, we rely on a procedure Recover() that recovers w_i from σ_w and $m_1, \dots, m_i$. (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 α_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-

Input: $w = w_1, \ldots, w_d$
// R_1 starts in its initial state σ_0.
$\mathsf{ins}_1([_0\,]_0, 0)$
for $i = 1, \ldots, d$
  // The state of R_1 here is σ_i.
  $send_1(m_i)$
  if $w_i = 1$ **then**
    $\mathsf{ins}_1([_i\,]_i$ just after $]_{i-1})$
  else
    $\mathsf{ins}_1([_i\,]_i$ just before $[_{i-1})$
  // The state of R_1 here is σ_{i+1}.
$send_1(m_{d+1})$
$\mathsf{ins}_1(*$ between $[_d$ and $]_d)$
$send_1(m_{d+2})$
for $i = 0, \ldots, d$
  $\mathsf{del}_1([_i)$
  $\mathsf{del}_1(]_i)$
if $D = 2(d+1)+1$ **then**
  $\mathsf{ins}_1(b, 0)$
  $\mathsf{del}_1(b)$
$send_1(m_{d+3})$
$\mathsf{read}_1 = *$
// The state of R_1 here is σ_w.

(a) Execution α_w, at the end of which the state of R_1 encodes w

Input: σ_w
// R_1 starts in its initial state σ_0,
// which does not depend on w.
$\mathsf{ins}_1([_0\,]_0, 0)$
for $i = 1, \ldots, d$
  // We now know σ_i,
  // so can generate m_i.
  $send_1(m_i)$
  $w_i \leftarrow$ Recover$(\sigma_w, m_1, \ldots, m_i)$
  if $w_i = 1$ **then**
    $\mathsf{ins}_1([_i\,]_i$ just after $]_{i-1})$
  else
    $\mathsf{ins}_1([_i\,]_i$ just before $[_{i-1})$
  // We now know σ_{i+1}.
output w

(b) Decoding w given σ_w

Input: $\sigma_w, m_1, \ldots, m_i$
// We perform the state transitions
// below on copies of the state
// machines, with R_2 starting in its
// initial state and R_1 in state σ_w.
// The replica state in the outer
// decoding procedure is unaffected.
for $j = 1, \ldots, i$
  $receive_2(m_j)$
  $receive_1(m_j)$
$\mathsf{read}_2 = \ldots\ [_{i-1}\,]_{i-1}\ \ldots$
$\mathsf{ins}_2(x$ between $[_{i-1}$ and $]_{i-1})$
$send_2(m_x)$
$receive_2(m_x)$
$receive_1(m_x)$
if $\mathsf{read}_1 = x*$
  return 1
else // $\mathsf{read}_1 = *x$
  return 0

(c) Recover$(\sigma_w, m_1, \ldots, m_i)$

Figure 4: Encoding and decoding procedures. Events are subscripted with the id of their replica.

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 σ_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 σ_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_3 is explained below). We start off with R_2 in its initial state and R_1 in state σ_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 $[_{i-1}\,]_{i-1} \in v_w^i$. Next, R_2 inserts element x between $[_{i-1}$ and $]_{i-1}$ 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 α_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 β_w^i of α_w of the form:

$$\begin{aligned}\beta_w^i = \alpha_w\ & receive_2(m_1)\, receive_1(m_1) \ldots receive_2(m_i)\, receive_1(m_i) \\ & do_2(\mathsf{read}_2, v_w^i)\, do_2(\mathsf{ins}_2(x, k_w^i), \_)\, send_2(m_x) \\ & receive_2(m_x)\, receive_1(m_x)\, do_1(\mathsf{read}_1, y_w^i),\end{aligned}$$

where k_w^i is the position at which R_2 inserts x into the list. In the following, we prove that the execution β_w^i 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 σ_w is R_1's state at the end of α_w, which allows Recover() to perform the same state transitions at R_1 that occur in β_w^i, without having access to the entire execution α_w that leads R_1 to state σ_w.

CLAIM 6. *Execution β_w^i is well-formed.*

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

CLAIM 7. *Execution β_w^i satisfies causal atomic broadcast.*

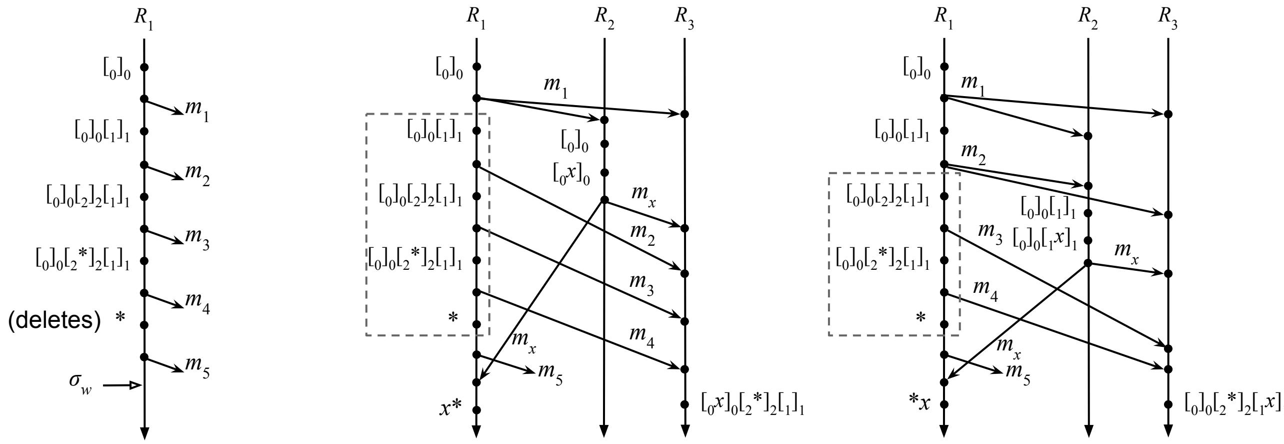
PROOF. Immediate from inspection of the message delivery order in β_w^i. $\square$

CLAIM 8. $[_{i-1}\,]_{i-1} \in v_w^i$.

PROOF. For $j = 1, \ldots, d+1$, let $e_j, f_j \in \alpha_w$ be the do events in which R_1 inserts $[_{j-1}$ and $]_{j-1}$ into the list. Let $r \in \beta_w^i$ be the do event at which R_2 reads v_w^i. Because $\mathcal{R}$ is a correct push-based protocol and β_w^i satisfies causal atomic broadcast, by Definition 13, β_w^i complies with some abstract execution $A = (H, \mathsf{vis}) \in \mathcal{A}_{\mathsf{weak}}$ such that $e_j \xrightarrow{\mathsf{vis}} r$ and $f_j \xrightarrow{\mathsf{vis}} r$ if and only if $j \le i$, and no del operation is visible to r. This holds because in β_w^i, 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 $v_w^i = \ldots\ [_{i-1}\,]_{i-1}\ \ldots$. $\square$

CLAIM 9. $y_w^i = x*$ *or* $y_w^i = *x$.

PROOF. Let $f \in \beta_w^i$ be the do event at which R_2 inserts x into the list, and $r \in \beta_w^i$ be the do event at which R_1 reads y_w^i. Because



(a) Encoding $w = 10$: execution α_w, at the end of which the state of R_1 is σ_w.

(b) Decoding $w_1 = 1$ given σ_w and m_1. (We do not show replicas receiving their own messages.)

(c) Decoding $w_2 = 0$ given σ_w, m_1 and m_2. (We do not show replicas receiving their own messages.)

Figure 5: Examples of the encoding and decoding procedures from Theorem 5 applied to $w = 10$. Figure 5(a) shows the execution α_w constructed by the encoding procedure, whose output is σ_w, the state of R_1 at the end of α_w. Figure 5(b) shows the first step of decoding w from σ_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 σ_w. The relative order of x and $*$ 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 α_w that depend on w_1.

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

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

In principle, the weak list specification allows R_1's read to order x and $*$ 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 $[_{i-1}\, x\,]_{i-1}$ and $*$ together, and therefore their order is fixed. Consider the following extension of β^i_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):

$$\gamma^i_w = \beta^i_w\, \mathit{receive}_3(m_1)\, \ldots\, \mathit{receive}_3(m_i)\, \mathit{receive}_3(m_x)$$
$$\mathit{receive}_3(m_{i+1})\, \ldots\, \mathit{receive}_3(m_{d+2})\, \mathit{do}_3(\mathsf{read}_3, z^i_w).$$

We show that the list z^i_w contains $*$ 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 γ^i_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 γ^i_w: w_*, the ins of $*$ by R_1; w_x, the ins of x by R_2; and r_z, the read by R_3, whose response is z^i_w. Because $\mathcal{R}$ is a correct push-based protocol and γ^i_w satisfies causal broadcast, by Definition 13, γ^i_w complies with some abstract execution $A = (H, \mathsf{vis}) \in \mathcal{A}_{\mathsf{weak}}$ such that for any $e, e' \in H$, $e' \xrightarrow{\mathsf{vis}} e$ if and only if $e' \xrightarrow{\mathsf{hb}} e$. Therefore, no del event is visible to w_*, w_x or r_z. Let lo be a list order that A is consistent with (Definition 8). We proceed to show that $(x, *) \in \mathsf{lo}$ 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_* establishes that $(]_{i-1}, *) \in \mathsf{lo}$ if and only if $w_i = 1$. The response of w_x establishes that $([_{i-1}, x) \in \mathsf{lo}$ and $(x,]_{i-1}) \in \mathsf{lo}$. It follows that $(x, *) \in \mathsf{lo}$ if and only if $w_i = 1$, since $[_{i-1}, x,]_{i-1}, * \in z^i_w$ and lo is total and transitive on $\{a \mid a \in \mathsf{rval}(r_z) = z^i_w\}$.

We conclude by noting that $y^i_w = x*$ or $y^i_w = *x$ (Claim 9); recall that y^i_w is the response to the read at R_1 performed by Recover(). Since $(x, *) \in \mathsf{lo}$ if and only if $w_i = 1$, then $y^i_w = x*$ 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 $R_i = (Q^i, M, \Sigma^i, \sigma^i_0, E, \Delta^i)$, and $\sigma \in \Sigma^i$, if $\Delta^i(\sigma, \mathit{receive}(m)) \neq \sigma$, 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 $\mathcal{R}' = \{R'_1, \ldots, R'_n\}$*

State	Event e	New state
$((r,s),m)$	$do(op,v)$	$((r',s),m')$, where $(r',m') = \Delta^i((r,m),e)$
$((r,s),m)$	$send(m)$	$((r',s),\perp)$, where $(r',\perp) = \Delta^i((r,m),e)$
$((r,s),m)$	$receive(m)$	$((r',s'),m')$, where if $\Delta^S((s,\perp),e) = (s^*,\perp)$, then $s' = s^*$ and $(r',m') = (r,m) = \Delta^i((r,m), receive(m))$; and if $\Delta^S((s,\perp),e) = (s^*,m^*)$, then $(s',\perp) = \Delta^S((s^*,m^*), send(m^*))$, $(r',m') = \Delta^i((\hat{r},\hat{m}), receive(m^*))$, where $(\hat{r},\hat{m}) = (r,m) = \Delta^i((r,m), receive(m))$

Figure 6: State machine of replica $R'_i \in \mathcal{R}'$ simulating replica $R_i = (Q^i, M, \Sigma^i, \sigma_0^i, E, \Sigma^i) \in \mathcal{R}$. The initial internal state is (q_0^i, q_0^S), where q_0^i and q_0^S are the initial internal states of R_i and of S, respectively.

that simulates $\mathcal{R}$ *in the following sense: (1) for any execution α' of $\mathcal{R}'$ that satisfies causal atomic broadcast, there exists an execution α of $\mathcal{R}$ such that $\alpha|^{do}_{R_i} = \alpha'|^{do}_{R'_i}$ 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 $(\_, q_0^S)$, where $(q_0^S, \perp)$ is the initial 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 m to the two state machines it maintains. (However, because m corresponds to a message sent by some R_j, 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 α' 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. $\square$

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 α_D of $\mathcal{R}'$ with D deletions such that: (1) the metadata overhead of some state σ of some replica $R' \in \mathcal{R}'$ is $\Omega(D)$; (2) α_D satisfies causal atomic broadcast; and (3) R' does not receive any message before σ. Because $\mathcal{R}'$ simulates $\mathcal{R}$, we have that $\sigma = ((q^R, q^S), \_)$, 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, $|(q^R, q^S)| = O(|q^R| + |q^S|) = O(|q^R|)$, 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 $\Omega(D)$. $\square$

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 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 [20] 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) [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 $O(D \lg k)$, 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

[1] Apache Wave. https://incubator.apache.org/wave/.

[2] H. Attiya, F. Ellen, and A. Morrison. Limitations of Highly-Available Eventually-Consistent Data Stores. In *PODC*, 2015.

[3] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on Causal Consistency. In *SIGMOD*, 2013.

[4] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *POPL*, 2014.

[5] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[6] J. Day-Richter. What's different about the new Google Docs: Conflict resolution. http://googledrive.blogspot.com/2010/09/whats-different-about-new-google-docs_22.html, 2010.

[7] J. Day-Richter. What's different about the new Google Docs: Making collaboration fast. http://googledrive.blogspot.com/2010/09/whats-different-about-new-google-docs.html, 2010.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[9] X. Défago, A. Schiper, and P. Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy And Survey. *ACM CSUR*, 36(4), 2004.

[10] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *SoCC*, 2013.

[11] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Closing The Performance Gap between Causal Consistency and Eventual Consistency. In *PaPEC*, 2014.

[12] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *SIGMOD*, 1989.

[13] A. Imine, M. Rusinowitch, G. Oster, and P. Molli. Formal Design and Verification of Operational Transformation Algorithms for Copies Convergence. *TCS*, 351(2), 2006.

[14] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7), 1978.

[15] B. Leuf and W. Cunningham. *The Wiki Way: Quick Collaboration on the Web*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[16] D. Li and R. Li. Preserving Operation Effects Relation in Group Editors. In *CSCW*, 2004.

[17] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual Consistency. *CACM*, 57(5), 2014.

[18] S. Martin, P. Urso, and S. Weiss. Scalable XML collaborative editing with undo. In *OTM*, 2010.

[19] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *UIST*, 1995.

[20] G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for P2P collaborative editing. In *CSCW*, 2006.

[21] N. Preguiça, J. M. Marqués, M. Shapiro, and M. Leția. A commutative replicated data type for cooperative editing. In *ICDCS*, 2009.

[22] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW*, 1996.

[23] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *JPDC*, 71(3), 2011.

[24] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A Comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, INRIA, Jan. 2011.

[25] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *SSS*, 2011.

[26] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *CSCW*, 1998.

[27] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM TOCHI*, 5(1), 1998.

[28] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*, 1995.

[29] W. Vogels. Eventually Consistent. *CACM*, 52(1), 2009.

[30] D. Wang, A. Mah, and S. Lassen. Google Wave Operational Transformation. https://wave-protocol.googlecode.com/hg/whitepapers/operational-transform/operational-transform.html, 2010.

[31] S. Weiss, P. Urso, and P. Molli. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *ICDCS*, 2009.

[32] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. Technical Report RR-8347, INRIA, 2013.

[33] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Middleware*, 2015.

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 TI 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 $\circ$, and can thus be added to A without breaking the conditions for TI trees).

Proof of Theorem 1. First, note that $\mathcal{R}^n_{\mathrm{rga}}$ 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 α be a concrete execution of $\mathcal{R}^n_{\mathrm{rga}}$ that satisfies causal broadcast. We define a replica order relation ro on the events in α by letting $e \xrightarrow{\mathsf{ro}} e'$ denote that e and e' are events by the same replica, and e precedes e' in α. Also, for two events $e, e' \in \alpha$, we define the delivery relation to indicate that the first message sent by $\mathsf{repl}(e)$ after e is received by $\mathsf{repl}(e')$ before e' :

$$
\begin{aligned}
(e \xrightarrow{\mathsf{deliv}} e') \iff \exists m :&\\
&(e \leq_{\mathsf{ro}} \mathit{send}(m)) \wedge (\mathit{receive}(m) \leq_{\mathsf{ro}} e')\\
&\qquad \wedge \neg(\exists m'.\, e \leq_{\mathsf{ro}} \mathit{send}(m')) <_{\mathsf{ro}} \mathit{send}(m).
\end{aligned}
$$

Without loss of generality, in the following we only consider executions α of the protocol that deliver all operations to all replicas: for each insertion or deletion operation e, and each replica R, there exists an event e' at R such that $e \xrightarrow{\mathsf{deliv}} e'$. If α 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 α.

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

(O1) α complies with H: for all replicas R, $H|_R = \alpha|_R^{do}$;

(O2) for $e, e' \in H$, $e' \xrightarrow{\mathsf{vis}} e$ if and only if $e' \xrightarrow{\mathsf{hb}} e$.

(O3) (H, vis) is an element of the strong list specification: $(H, \mathsf{vis}) \in \mathcal{A}_{\mathsf{strong}}$.

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

CLAIM 12. *(H, vis) is an abstract execution.*

PROOF. Following Def. 4, we need to check that vis is acyclic. This is obvious because either $e \xrightarrow{\mathsf{deliv}} e'$ or $e \xrightarrow{\mathsf{ro}} e'$ implies that $e <_\alpha e'$, thus $e \xrightarrow{\mathsf{vis}} e'$ also implies $e <_\alpha e'$, thus it is acyclic. Condition (1) is satisfied because $\mathsf{ro} \subseteq \mathsf{vis}$. Condition (2) is satisfied because $e \xrightarrow{\mathsf{vis}} e'$ implies $e <_\alpha e'$ and thus $e <_H e'$. Condition (3) is satisfied because vis is transitive by definition. □

Clearly, obligation (O1) is satisfied: α complies with H since the order of events in H matches their order in α. Also, obligation (O2) is satisfied by definition of vis and 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 $a \in \mathsf{elems}(H)$, 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 α, and let $(N', \_, \_)$ be state of the replica $\mathsf{repl}(e)$ after executing e. Then N' contains all nodes that were inserted by e or insertion operations visible to e: $N' = \{(a, t_a, p_a) \mid e_a \leq_{\mathsf{vis}} e\}$.*

PROOF. By induction over α 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. $\mathsf{vis}^{-1}(e)$ contains the union of visible operations $\mathsf{vis}^{-1}(e')$ 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 α, and let $(\_, T', \_)$ be the state of the replica $\mathsf{repl}(e)$ after executing e. Then, T' contains all elements that were deleted by e or deletion operations visible to e: $T' = \{a \mid do(\mathsf{del}(a)) \leq_{\mathsf{vis}} e\}$.*

PROOF. By induction over α 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. $\mathsf{vis}^{-1}(e)$ contains the union of visible deletions in $\mathsf{vis}^{-1}(e')$ of the sending event e' and the predecessor event on the same replica. Symmetrically, T is updated to contain the delivered tombstones, 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 $(N, \_, \_)$ 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 $(a, t_a, \_) \xrightarrow{\mathsf{pa}} (b, t_b, \_)$ implies $e_a \xrightarrow{\mathsf{vis}} e_b$ 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) is in N, then $e_a \xrightarrow{\mathsf{vis}} e_b \xrightarrow{\mathsf{vis}} e$, thus by transitivity of vis also $e_a \xrightarrow{\mathsf{vis}} e$, thus $(a, t_a, \_) \in N$. $\square$

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

LEMMA 16. lo *defines a total order on* $\mathsf{elems}(H)$.

PROOF. *Irreflexive.* All N are TI trees by Lemma 15, and thus contain no duplicate elements. *Total.* Let $a, b \in \mathsf{elems}(H)$, inserted by events e_a and e_b, respectively. Since e_a enables a send event, $\mathsf{repl}(e_a)$ must eventually send a message containing $(a, \_, \_)$, and $\mathsf{repl}(e_b)$ must receive it (recall that we only consider executions where every message is delivered), either before or after e_b. After both of those, the replica state on $\mathsf{repl}(e_b)$ contains in N nodes for both a and b, thus $s(N)$ orders them. *Transitive.* Let $a, b, c \in \mathsf{elems}(H)$. 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 $N_1 \subseteq N_3$ and $N_2 \subseteq N_3$. By Lemma 3 and Lemma 15, in $s(N_3)$, a appears before b and b before c. Thus a appears before c in $s(N_3)$. $\square$

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 $w = a_1 \dots a_n$ 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

$$\begin{aligned} s_N &= \sum_i (s_i + O(\lg k)) = O(\lg k)(\sum_i s_i) \\ s_T &= d \cdot O(\lg k) \\ s_A &\leq s_N \\ s_K &\leq s_T \end{aligned}$$

And we get

$$\begin{aligned} \frac{s_N + s_T + s_A + s_K}{s_w} &\leq \frac{2s_N + 2s_T}{s_w} \\ &\leq 2\frac{O(\lg k)(\sum_i s_i) + d \cdot O(\lg k)}{(\sum_i s_i)} \\ &\leq 2(O(\lg k) + d \cdot O(\lg k)) \\ &= O((\lg k)(1 + d)). \end{aligned}$$

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 $\ell_N(p) \in (U \times C)^*$ of element/timestamp pairs, as follows:

$$\ell_N(p) = (p.a, p.t)\ \ell_N(c_1)\ \dots\ \ell_N(c_n)$$

where $\{c_1, \dots, c_n\}$ 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 $p = \circ$, that is, naming the root as a parent, and let $\circ.a = \epsilon$, and $\circ.t = 0$ for some timestamp 0 that is smaller than any other timestamp. So, $\ell_N(\circ)$ 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

$$\mathit{splice}(w, a', t', p') = w_1\ (a, p')\ w_2\ (a', t')\ w_3$$

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: $\mathit{splice}(\ell_N(\circ), a', t', p') = \ell_{N'}(\circ)$.*

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 $p \in N \cup \{\circ\}$ that are ancestors of (a', t', p') in N', we have $\mathit{splice}(\ell_N(p), a', t', p') = \ell_{N'}(p)$.*

PROOF. We use induction over the number of descendants of p in N', and do a case distinction.

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

$$\begin{aligned} &\mathit{splice}((\ell_N(p), a', t', p') \\ &= \mathit{splice}(((p.a, p.t)\ \ell_N(c_1)\ \dots\ \ell_N(c_n), a', t', p') \\ &= \mathit{splice}(((p.a, p')\ w_2\ w_3, a', t', p') \end{aligned}$$

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 *splice* definition, and we get

$$= (p.a, p')\ w_2\ (a', t')\ w_3$$
$$= (p.a, p.t)\ \ell_N(c_1) \ldots \ell_N(c_i)\ (a', t')\ \ell_N(c_{i+1}) \ldots \ell_N(c_n)$$

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

$$= \ell_{N'}(p)$$

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

$$\ell_{N'}(p) = (p.a, p.t)\ \ell_{N'}(c_1) \ldots \ell_{N'}(c_n)$$

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'. Then

$$= (p.a, p.t)\ \ell_N(c_1) \ldots \ell_N(c_{i-1})\ \ell_{N'}(c_i)\ \ell_N(c_{i+1}) \ldots \ell_N(c_n)$$

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 $\ell_{N'}(c_i)$, we get

$$= (p.a, p.t)\ \ell_N(c_1) \ldots \ell_N(c_{i-1})$$
$$\mathit{splice}(\ell_N(c_i), a', t', p') \quad \ell_N(c_{i+1}) \ldots \ell_N(c_n)$$

and expanding the pattern in *splice* gives us

$$= (p.a, p.t)\ \ell_N(c_1) \ldots \ell_N(c_{i-1})$$
$$w_1\ (a, p')\ w_2\ (a', t')\ w_3 \quad \ell_N(c_{i+1}) \ldots \ell_N(c_n)$$

where w_1, w_2, w_3 satisfy $\ell_N(c_i) = w_1\ (a, p')\ w_2\ w_3$, and (1) all timestamps appearing in w_2 are larger than t', and (2) the first timestamp appearing in w_3 is smaller than t', or w_3 is empty. But now we can regroup this sequence, obtaining

$$= (p.a, p.t)\ w_1'\ (a, p')\ w_2\ (a', t')\ w_3'$$

where we define $w_1' = \ell_N(c_1) \ldots \ell_N(c_{i-1})\ w_1$, and we define $w_3' = w_3\ \ell_N(c_{i+1}) \ldots \ell_N(c_n)$. Note that either w_3' is empty, or its first timestamp is smaller than t' (because if not empty, the first element of $\ell_N(c_{i+1})$ is $c_{i+1}.t$ which is smaller than $c_i.t$, which is smaller than t' because it is an ancestor of t'). Therefore,

$$= \mathit{splice}((\ell_N(p), a', t', p')$$

□

B. STRICTLY WRITE-PROPAGATING PROTOCOLS ARE PUSH-BASED

Many highly available eventually consistent protocols [3,4,8,10, 11,17,25,32] are *write-propagating* [2]: they have *invisible reads*, which complete without changing the replica state, and *op-driven messages*, which means that replicas generate messages only as a result of user operations and not in response to a received message. We repeat the definition here, adapted to our model:

DEFINITION 15. *A protocol $\mathcal{R}$ is* write-propagating *if it satisfies eventual consistency and the following hold for every replica $R = (Q, M, \Sigma, \sigma_0, E, \Delta)$:*

- *User reads do not change the state of a replica: if $\Delta(\sigma_1, do(op, v)) = \sigma_2$ and $\mathsf{op}(e) = \mathsf{read}$, then $\sigma_1 = \sigma_2$.*
- *$\mathcal{R}$ generates messages only as a result of user operations, and not in response to received messages: R does not have a message pending in σ_0, and if $\sigma_2 = \Delta(\sigma_1, receive(m))$ and R does not have a message pending in σ_1, then R does not have a message pending in σ_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 $\mathcal{R}$ is* strictly write-propagating *if it is a write-propagating protocol and for every replica $R = (Q, M, \Sigma, \sigma_0, E, \Delta)$, if $\Delta(\sigma_1, send(m)) = \sigma_2$, then $do(op, v)$ is an enabled transition in σ_1 if and only if $do(op, v)$ is an enabled transition in σ_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 $\mathcal{R}$ be an eventually consistent write-propagating protocol satisfying the weak (respectively, strong) list specification. Then $\mathcal{R}$ is a push-based protocol.*

PROOF. We show that $\mathcal{R}$ 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 α be a well-formed execution of $\mathcal{R}$ and let e be an event in α. Then the following sequences of events are well-formed executions of $\mathcal{R}$:*

1. *β, the subsequence of α consisting of all events e' such that $e \overset{\mathsf{hb}}{\not\longrightarrow} e'$.*
2. *γ, the subsequence of α consisting of all events e' such that $e' \xrightarrow{\mathsf{hb}} e$.*

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

PROPOSITION 21. *Let α be an execution of $\mathcal{R}$. Let $A = (H, \mathsf{vis})$ an abstract execution that α complies with. Let o be a $do(op, v)$ event and e be a $do(\mathsf{ins}(a, \_), \_)$ event such that $a \in v$. Then $e \xrightarrow{\mathsf{hb}} o$.*

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

LEMMA 22. *Let α be of $\mathcal{R}$ and $e \in \alpha$. Then (1) if $\mathsf{op}(e) = \mathsf{ins}(a, \_)$, $\mathsf{repl}(e)$ has a message pending after e; and (2) if $\mathsf{op}(e) = \mathsf{del}(a)$ and there does not exist $e' \xrightarrow{\mathsf{hb}} e$ with $\mathsf{op}(e') = \mathsf{del}(a)$, then $\mathsf{repl}(e)$ has a message pending after e.*

PROOF. Suppose the claim is false. Let σ be the state of $R = \mathsf{repl}(e)$ after e (in which R does not have a message pending). Let α_1 be the subsequence of α consisting of all events e' such that $e' \xrightarrow{\mathsf{hb}} e$. By Proposition 20, α_1 is a well-formed execution, and $\alpha_1|R$ if a prefix of $\alpha|R$. Thus, R's state at the end of α_1 is also σ.

Let $\beta = \alpha_1\,\alpha_2$ be an execution of $\mathcal{R}$ obtained by appending *receive* events to α_1 in some arbitrary order, so that in β every replica receives every message sent by another replica in α_1. Because $\mathcal{R}$ has op-driven messages, it follows that R does not have a message pending at the end of β. Moreover, no replica $R' \neq R$ has a message pending at the end of β: If there are no events at R' in α_1, then it does not have a message pending at the end of α_1. Otherwise, consider the last event e' at R' in α_1. Then $e' \xrightarrow{\mathsf{hb}} e$, by definition of α_1, and so R' sent a message at or after e'. But e' is the last event at R', so e' is a *send* event.

Consider the infinite execution of $\mathcal{R}$, $\beta_\infty = \beta\, r_1\, r_2 \ldots$, where the r_i events are reads at some $R' \neq R$. Let $A = (H, \mathsf{vis})$ be an abstract execution satisfying eventual visibility that β_∞ complies with. Then there exists some r_j such that $e \xrightarrow{\mathsf{vis}} r_j$. We now consider the two possible cases.

(1) $\mathsf{op}(e) = \mathsf{ins}(a, \_)$ Because $e \overset{\mathsf{hb}}{\not\longrightarrow} r_j$, by Proposition 21, $a \notin \mathsf{rval}(r_j)$. It follows from the weak list specification that there exists a $\mathsf{del}(a)$ event, $d \in H$, such that $d \xrightarrow{\mathsf{vis}} r_j$. It follows from our assumption that users delete only elements that appear in the response of a preceding operation on the same replica that $a \in \mathsf{rval}(e')$ for some $e' \xrightarrow{\mathsf{hb}} d$. By Proposition 21, $e \xrightarrow{\mathsf{hb}} e'$. But this is a contradiction, since $e' \xrightarrow{\mathsf{hb}} e$.

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

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

PROOF. It is immediate that there exists an abstract execution $A = (H, \mathsf{vis})$ that α complies with such that for all $e', e \in H$, $e' \xrightarrow{\mathsf{vis}} e$ if and only if $e' \xrightarrow{\mathsf{hb}} e$. To show that $A \in \mathcal{A}_{\mathsf{weak}}$ (respectively, $A \in \mathcal{A}_{\mathsf{strong}}$), 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 *do* event $e \in \alpha$, $\mathsf{rval}(e) = L(e, \mathsf{vis})$, where

$$L(e, \mathsf{vis}) = \{a \mid (do(\mathsf{ins}(a, \_), \_) \leq_{\mathsf{vis}} e) \wedge \neg(do(\mathsf{del}(a), \_) \leq_{\mathsf{vis}} e)\}.$$

Let e be an *do* event in α, let $R = \mathsf{repl}(e)$, and let σ be the state of R when the e transition occurs. Without loss of generality, R does not have a message pending in σ. (If R has a message m pending in σ, we can first send it; because R is strictly write-propagating, e will remain an enabled transition.) Let α' be the subsequence of α consisting of all events e' such that $e' \xrightarrow{\mathsf{hb}} e$. By Proposition 20, α' is a well-formed execution of $\mathcal{R}$, and since α satisfies causal broadcast, so does α'. Therefore, any message sent in α' is received by R before e. Let β be an extension of α' that satisfied causal broadcast, in which every replica receives every message sent by another replica in α'. Because $\mathcal{R}$ has op-driven messages, no replica has a message pending at the end of β. Moreover, by construction of α', R does not receive any messages after e in β.

$L(e, \mathsf{vis}) \subseteq \mathsf{rval}(e)$: Consider $a \in L(e, \mathsf{vis})$. Then $\mathsf{del}(a) \overset{\mathsf{hb}}{\not\longrightarrow} e$, by definition of $L(e, \mathsf{vis})$, and so $\mathsf{del}(a) \notin \beta$, by construction of β. Consider the execution β' of $\mathcal{R}$ obtained by removing e and executing infinitely many reads $r_1, \ldots$ at R. It follows from eventual consistency and the fact that $\mathcal{R}$ satisfies the weak (respectively, strong) list specification, that there exists some abstract execution satisfying eventual visibility $A' = (H', \mathsf{vis}') \in \mathcal{A}_{\mathsf{weak}}$ (respectively, $A' \in \mathcal{A}_{\mathsf{strong}}$) that β' complies with. Therefore, for some r_i, $\mathsf{ins}(a, \_) \xrightarrow{vis'} r_i$. Since $\mathsf{del}(a) \notin H'$, it follows that $a \in \mathsf{rval}(r_i)$. Since $\mathcal{R}$ has invisible reads, the state of R at r_i is σ, the state in which the e transition occurs. It follows that $a \in \mathsf{rval}(e)$.

$L(e, \mathsf{vis}) \supseteq \mathsf{rval}(e)$: Consider $a \in \mathsf{rval}(e)$. By Proposition 21, $\mathsf{ins}(a, \_) \xrightarrow{\mathsf{hb}} e$, and so $\mathsf{ins}(a, \_) \xrightarrow{\mathsf{vis}} e$. Suppose that $\mathsf{del}(a) \xrightarrow{\mathsf{hb}} e$, and consider the execution β' of $\mathcal{R}$ obtained by removing e and executing infinitely many reads $r_1, \ldots$ at R. It follows from eventual consistency and the fact that $\mathcal{R}$ satisfies the weak (respectively, strong) list specification, that there exists some abstract execution satisfying eventual visibility $A' = (H', \mathsf{vis}') \in \mathcal{A}_{\mathsf{weak}}$ (respectively, $A' \in \mathcal{A}_{\mathsf{strong}}$) that β' complies with. Therefore, for some r_i, $\mathsf{del}(a) \xrightarrow{vis'} r_i$. It follows that $a \notin \mathsf{rval}(r_i)$. Since $\mathcal{R}$ has invisible reads, the state of R at r_i is σ, the state in which the e transition occurs. Therefore, $a \notin \mathsf{rval}(e)$, which is impossible. Thus, $a \in L(e, \mathsf{vis})$. $\square$

This concludes the proof.