Theoretical Computer Science 855 (2021) 141–160

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Specification and space complexity of collaborative text editing

Hagit Attiya [a,*], Sebastian Burckhardt [b], Alexey Gotsman [c], Adam Morrison [d], Hongseok Yang [e], Marek Zawirski [f]

[a] Technion, Israel
[b] Microsoft Research, United States of America
[c] IMDEA Software Institute, Spain
[d] Tel Aviv University, Israel
[e] KAIST, Republic of Korea
[f] Google, Switzerland

ARTICLE INFO

Article history:
Received 11 March 2020
Received in revised form 16 October 2020
Accepted 26 November 2020
Available online 9 December 2020
Communicated by L. Christoph

Keywords:
Collaborative text editing
Specification
Metadata lower bound
Eventual consistency

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 abstract, high-level specification of their desired behavior, which is decoupled from their actual implementation. Several of these protocols have been shown not to satisfy even basic expectations. This paper provides a precise specification of a replicated abstract *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, in particular CRDT and OT protocols, and we show that one of these protocols almost matches the bound. The result holds for peer-to-peer protocols, even if the network guarantees causal atomic broadcast. The result also holds for the metadata cost at the clients in client/server protocols.[1]

1. Introduction

Collaborative text editing systems, like Google Docs [10,9], Apache Wave [1], or wikis [17], allow users at multiple sites to concurrently edit the same document. To achieve high responsiveness and availability, such systems often replicate the

* Corresponding author.
E-mail address: hagit@cs.technion.ac.il (H. Attiya).

[1] A preliminary version of the paper appeared in [4]. The current version adds Section 7 and Appendix A, includes proof for Section 4, fixes mistakes, presentation issues and typos, and expands and updates the discussion of related work.

https://doi.org/10.1016/j.tcs.2020.11.046
0304-3975/

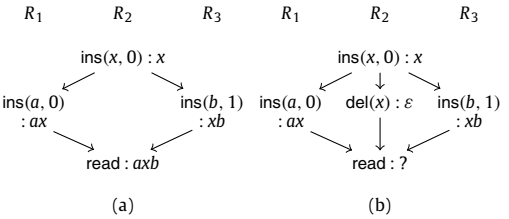


Fig. 1. Example scenarios of collaborative text editing. Events are presented in format "operation: return value". An arrow from an event e' to an event e expresses that the effects of e' get incorporated at e's replica before e executes.

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 [36,35].

Fig. 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 this point, consider the scenario in Fig. 1(b) (also known as the TP2 puzzle [21]), 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 [27]; e.g., we demonstrate in Appendix A that this is the case in the Jupiter protocol [20] used in public collaboration systems [37].

Users would like to have *highly available* protocols, which respond to their operations immediately, without performing any communication. There have been a number of proposals of highly available collaborative editing protocols, using techniques such as *operation transformations* (OT) [12,25,29,30] and *conflict-free replicated data types* (also known as CRDTs) [23,39,26]. It is challenging to specify the desired behavior of collaborative editing protocols, without referring to the actual implementation, in particular, for identifying the *visible* operations that should be reflected in the responses of operations. Abstracting away implementation details is essential for studying inherent properties and limitations. Instead, existing specifications [30,18] refer to implementation details, e.g., messages sent and received; this does not provide a common ground on which to compare different implementation approaches for the same specification [32]. In addition, several of the protocols have been shown not to satisfy even the basic expectation of eventual consistency [15].

We introduce an implementation-independent specification of a replicated list object. The list object 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 of list elements observed by different clients are consistent. This includes transitive consequences of orderings over subsequently deleted elements, such as x in Fig. 1(b), and thus, the strong specification disallows the response ba for the read in this figure. The *weak* specification does not take transitivity into account, thus allowing the read in Fig. 1(b) to return either ba or ab. We show that both of these specifications ensure eventual consistency.

We prove that the strong specification is correctly implemented by a variant of the RGA (Replicated Growable Array) protocol [26], which is in the style of replicated data types [23] (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 Fig. 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 Fig. 1(b)).

The simplicity of handling deletions via tombstones in the RGA protocol comes with a high space overhead. More precisely, the *metadata overhead* [6] 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 [39] 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.[2]

Other CRDT protocols do not keep tombstones, e.g., Treedoc [23], Logoot [39] or WOOT [22], but the replica state contains metadata, e.g., labels, that grows linearly in the number of deletions. OT protocols [12] pay metadata overhead by logging unacknowledged updates in each replica.

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

Our main result is that this overhead is indeed, in some sense, inherent. We prove that any push-based protocol that implements the weak list specification for $n \geq 3$ replicas incurs a metadata overhead of $\Omega(D)$, where D is the number of deletions. In a *push-based* protocol, 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 and it includes both CRDT and OT protocols. The lower bound holds even if the network guarantees causal atomic broadcast [11].

We first establish our lower bound for the peer-to-peer model. *Client/server* protocols attempt to save space on replicas by keeping it on a central server. Replicas communicate only with the server and not directly with each other, and so the server does more than merely relay messages between replicas. Using the fact that the lower bound 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 $\Omega(D)$. This shows that relying on a central server does not reduce the metadata overhead at the clients.

We prove our lower bound using an inductive information-theoretic argument, which we consider to be the novel technical contribution of this paper. 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.

The class of *push-based protocols*, to which our metadata overhead lower bound proof applies, is specified with low-level properties that partially restrict implementation details of the protocol. We show, however, that under a weaker network model (defined in Section 7), if a protocol has *invisible reads* (which do not change the state of the replica), then being push-based follows from guaranteeing the higher-level property of eventual consistency. Hence, our lower bound also applies to protocols satisfying the latter property.

2. System model

We are concerned with highly available implementations of a replicated object [5,6], 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*, $\Sigma \subseteq Q \times (M \cup \{\bot\})$ 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 replica can execute any user 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 assume that the response of a user operation depends only on the replica's internal state: for every $(q, m) \in \Sigma$ and $(q, m') \in \Sigma$, if $do(op, v)$ is enabled in state (q, m), then $do(op, v)$ is enabled in state (q, m').

We require that a $send(m)$ event is enabled in state σ if and only if $\sigma = (q, m)$ for some replica q and $m \neq \bot$, and in this case $\Delta((q, m), send(m)) = (q, \bot)$. 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[3]; replicas can implement point-to-point communication by ignoring messages for which they are not the intended recipient.

[3] The latter is used to support atomic broadcast [11], defined later.

Executions. An *execution* of a protocol $\mathcal{R}$ is a (possibly infinite) sequence of events occurring at the replicas in $\mathcal{R}$.[4] 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 $e' \in \alpha$ [16] (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 $f \xrightarrow{\mathsf{hb}} e'$.

Network model. To ensure that every operation *eventually* propagates to all the replicas, we require that the network does not remain indefinitely disconnected. 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 [29,26]. We model this by considering only executions that satisfy *causal broadcast* [7]:

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* [11] model, which delivers all messages to all replicas in the exact same order.

Definition 3. An execution α of a 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 [7]; 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.

[4] 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.

- 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 list and skip the deletion if the deleted element does not appear in the list.

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. [6] for specifying replicated data types [28]. 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 [29], here we consider only *causal* abstract executions, where the visibility relation is transitive. In the following definition, a sequence H of *do* events is 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 representation.

Definition 4. A *causal abstract execution* is a pair (H, vis), where H is a sequence of *do* events, 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 $e_1 \xrightarrow{\mathsf{vis}} e_3$).

Fig. 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. The visibility relation abstracts away message-delivery information about the execution, by only postulating the existence of a consistent (acyclic) way to relate events.

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 $\mathcal{R}$ that satisfies causal broadcast complies with some abstract execution $A \in \mathcal{S}$.

3.2. Specifying the list object

We present two list specifications: strong and weak. The *strong* specification ensures that orderings of list elements observed by different clients are consistent, including transitive consequences of orderings over subsequently deleted elements. The specification thus disallows the response *ba* for the read in Fig. 1(b). The *weak* specification does not take transitivity into account, and thus allows 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 some relation $\mathsf{lo} \subseteq \mathsf{elems}(A) \times \mathsf{elems}(A)$, called the *list order*, such that:

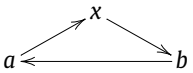
1. Each event $e = do(op, w) \in H$ returns a sequence of elements $w = a_0 \ldots a_{n-1}$, where $a_i \in \mathsf{elems}(A)$, such that
   (a) w contains exactly the elements visible to e that have been inserted, but not deleted:

$$\forall a.\, a \in w \iff (do(\mathsf{ins}(a, \_), \_) \leq_{\mathsf{vis}} e) \wedge \neg(do(\mathsf{del}(a), \_) \leq_{\mathsf{vis}} e).$$

   (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 Fig. 1(a) and the one in Fig. 1(b) with the read returning *ab*; this is justified by the list order $a \to x \to b$. (See the picture on the right.) On the other hand, the specification is not satisfied by the execution in Fig. 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 [26]. However, some protocols, such as Jupiter [20], provide weaker guarantees and, in particular, allow the outcome *ba* in Fig. 1(b) [27]. We therefore introduce the following weak list specification, to which our lower bound result applies (Section 6).



Definition 8. An abstract execution $A = (H, \mathsf{vis})$ belongs to the *weak list specification* $\mathcal{A}_{\mathsf{weak}}$ if and only if there is some 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 Fig. 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 $x \to b$. Since the conference publication of this work, it has been proved that Jupiter indeed satisfies the weak list specification [38].

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 [36,35]. 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*, which (informally) requires that the effects of every update operation eventually propagate to all replicas.

Definition 9. An abstract execution $A = (H, \mathsf{vis})$ satisfies *eventual visibility* if for every update event $e \in H$, there are only finitely many events $e' \in H$ such that $\neg(e \xrightarrow{\mathsf{vis}} e')$.

Definition 10. A protocol $\mathcal{R}$ satisfying the weak (respectively, strong) list specification *guarantees eventual visibility* if every execution α of $\mathcal{R}$ that satisfies causal broadcast 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 some encoding; we denote the size of x by $|x|$. The *metadata overhead* [6] 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 *([6]).* 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.

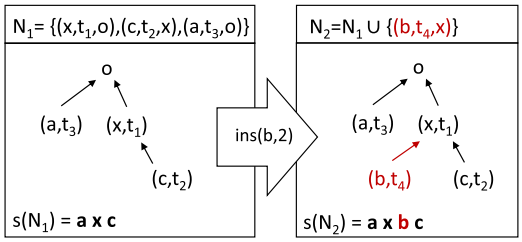


Fig. 2. Illustration of TI trees. Each box shows a tree, with the set of nodes that define it, its graphical representation, and the sequence of elements it denotes. The tree on the right results from an insert operation for element b at position 2. The order on the timestamps is $t_1 < t_2 < t_3 < t_4$.

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 [26], 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.

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 $p \in (U \cup \{\circ\})$ is either the parent node (identified by its symbol) 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 $n \xrightarrow{\text{pa}} n'$.

Definition 12. A set of nodes N is a *TI tree* if
(1) timestamps or symbols uniquely identify nodes:

$$\forall n, n' \in N.\ (n.t = n'.t \vee n.a = n'.a) \implies n = n';$$

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

Fig. 2 shows examples of TI trees and illustrates the read and insert operations explained below.

Read. To read the list, we traverse the tree N by depth-first search, starting at the root. We assemble the visited elements into a sequence $s(N)$ using prefix order (the parent precedes its children) and visit the children in decreasing timestamp order.

Insert. To insert a new element a at position k into the list, let $s(N) = a_0 \ldots a_{n-1}$ and pick a new timestamp t that is larger than any of the timestamps appearing in N. Then let p be the element to the left of the insertion position: $p = a_{k-1}$ (if $k > 0$) or $p = \circ$ (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 $\mathcal{R}^n_{\text{rga}}$ for n replicas. Each replica stores a TI tree, as well as a set of elements that represent *tombstones*, used to handle deletions (Section 1). Insertions and deletions are recorded in a send buffer, which is periodically transmitted to other replicas by causal broadcast.

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

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

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

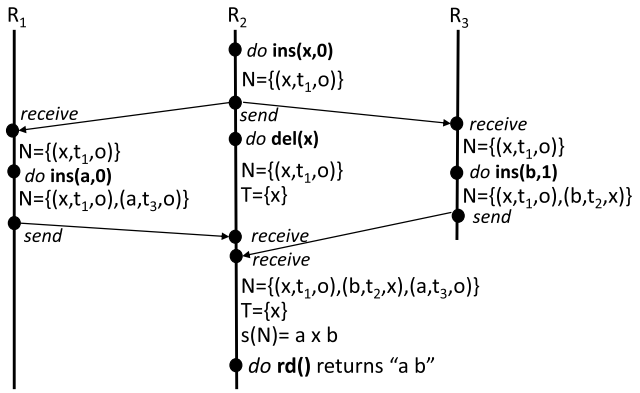


Fig. 3. Illustration of an RGA execution, with time proceeding from top to bottom. Bullets show the transitions of the replicas R_1, R_2, and R_3, and in some places we indicate the current state of the tree N and the tombstone set T. The order on the timestamps is $t_1 < t_2 < t_3$.

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 a TI tree after adding A_m.

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

4.3. Correctness and complexity

The following theorems state the correctness and asymptotic complexity bounds of RGA.

Theorem 1. *The protocol $\mathcal{R}^n_{\text{rga}}$ satisfies the strong list specification and guarantees eventual visibility.*

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

Proof. Since timestamps grow with the number of operations performed, 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

$$s_N = \sum_i (s_i + O(\lg k)) \le O(\lg k)(\sum_i (s_i + 1))$$
$$= O(\lg k)(\sum_i s_i) \ \text{(by } s_i + 1 \le 2s_i)$$
$$s_T = D \cdot O(\lg k)$$
$$s_A \le s_N$$
$$s_K \le s_T$$

And we get

$$\frac{s_N + s_T + s_A + s_K}{s_w} \le \frac{2s_N + 2s_T}{s_w}$$
$$\le 2\frac{O(\lg k)(\sum_i s_i) + D \cdot O(\lg k)}{(\sum_i s_i)} \le 2(O(\lg k) + D \cdot O(\lg k)) = O((\lg k)(1 + D)). \quad \square$$

We now proceed to prove Theorem 1.

Let α be a concrete execution of $\mathcal{R}^n_{\text{rga}}$ that satisfies causal broadcast. Without loss of generality, we assume that in α all operations are delivered to all replicas: for each insertion or deletion operation e, and each replica R, there exists an event e' at R such that $e \xrightarrow{\text{deliv}} e'$. If α is infinite, this already follows from our definition of a sufficiently connected network. Otherwise, we can simply append some additional send and/or receive events to α.

Let H be the subsequence of all *do* events in α; then α complies with H. Let vis be defined as follows: for $e, e' \in H$, $e' \xrightarrow{\text{vis}} e$ if and only if $e' \xrightarrow{\text{hb}} e$. It is easy to see that $A = (H, \text{vis})$ is an abstract execution satisfying eventual visibility. We now show that A satisfies the strong list specification. To this end, we prove the conditions of Definition 7 in the following order: (1c), (1a), (1b), (2).

Condition (1c) follows directly from the properties of the TI 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 lemmas do just that, and together imply condition (1a).

For each element $a \in \text{elems}(A)$, let e_a be the event that inserted it into the list, and let (a, t_a, p_a) be the tuple constructed during insertion.

Lemma 3. *Let e be an event in α, and let $(N', \_, \_)$ be state of the replica $\text{repl}(e)$ after executing e. Then N' contains all nodes that were inserted by e or insertion operations visible to e: $N' = \{(a, t_a, p_a) \mid e_a \le_{\text{vis}} e\}$.*

Proof. By induction over α and considering the type of event e.

If e is an 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.

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

In all other cases, neither the visible insertion operations nor N are updated. $\square$

Lemma 4. *Let e be an event of α, and let $(\_, T', \_)$ be the state of the replica $\text{repl}(e)$ after executing e. Then, T' contains all elements that were deleted by e or deletion operations visible to e: $T' = \{a \mid do(\text{del}(a)) \le_{\text{vis}} e\}$.*

The proof of the lemma is analogous to that of Lemma 3.

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

Lemma 5. *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 $(a, t_a, \_) \xrightarrow{\text{pa}} (b, t_b, \_)$ implies $e_a \xrightarrow{\text{vis}} e_b$ by Lemma 3, 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 3, 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 $(a, t_a, \_) \in N$. □

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 Definition 7:

Lemma 6. *Let A, B be two TI trees such that $A \subseteq B$. Then $s(A)$ is a subsequence of $s(B)$.*

Proof. This is easy to see for the special case where $B = A \uplus \{(a, t, p)\}$ for some a, t, p. Indeed, because A and B are TI trees, a must have no children in B, and 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 the above special case we get to the general case by induction, adding one node at a time: there always exists at least one node in $B \setminus A$ whose parent is in A or is $\circ$, and can thus be added to A without breaking the conditions for TI trees. □

Lemma 7. lo *defines a total order on* $\mathsf{elems}(A)$.

Proof. *Irreflexive.* All N are TI trees by Lemma 5, 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 6 and Lemma 5, in $s(N_3)$, a appears before b and b before c. Thus a appears before c in $s(N_3)$. □

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

5. Push-based protocols

Our lower bound results hold for *push-based* protocols, a class of protocols that contains the protocols of several collaborative editing systems [20,23,26,29], 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 push-based protocols assuming that the network provides causal broadcast, because when this is not the case, a protocol may need to delay merging arriving updates to enforce causality. Formally, we require that, in a push-based protocol, every operation observe all operations that happen before it, that list insertions always generate a message, and that a deletion—which, unlike an insertion, may not be unique—generates a message if it does not already observe another deletion of the same element.

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

- For any execution α of $\mathcal{R}$ satisfying causal broadcast and $e = do(\mathsf{ins}(a, \_), \_) \in \alpha$, replica $\mathsf{repl}(e)$ has a message pending after e.
- For any execution α of $\mathcal{R}$ satisfying causal broadcast 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 $\mathcal{R}$ satisfying causal broadcast 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 $\forall e', e \in H.\, e' \xrightarrow{\mathsf{vis}} e \iff e' \xrightarrow{\mathsf{hb}} e$.

Our proof of Theorem 1 also establishes that the RGA protocol is push-based. More generally, the class of push-based protocols contains both *op-based* protocols [6], in which a message carries a description of the latest operations that the sender has performed (e.g., RGA), and *state-based* protocols [6], in which a message describes all operations the sender knows about (i.e., its state). An OT protocol satisfying one of our specifications will also be push-based, as OT protocols propagate updates to remote replicas immediately [3,20].

6. Lower bounds on metadata overhead

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

```
Input:@ w = w_1, ..., w_d
// R_1 starts in its initial
// state σ_0.
ins_1([_0 ]_0, 0)
for i = 1, ..., d
   // The state of R_1
   // here is σ_i.
   send_1(m_i)
   if w_i = 1 then
      ins_1([_i ]_i just after ]_{i-1})
   else
      ins_1([_i ]_i just before [_{i-1})
   // The state of R_1
   // here is σ_{i+1}.
send_1(m_{d+1})
ins_1(∗ between [_d and ]_d)
send_1(m_{d+2})
for i = 0, ..., d
   del_1([_i)
   del_1(]_i)
if D = 2(d + 1) + 1 then
   ins_1(b, 0)
   del_1(b)
send_1(m_{d+3})
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.
ins_1([_0 ]_0, 0)
for i = 1, ..., d
   // We now know σ_i,
   // so can generate m_i.
   send_1(m_i)
   w_i ← Recover(σ_w, m_1, ..., m_i)
   if w_i = 1 then
      ins_1([_i ]_i just after ]_{i-1})
   else
      ins_1([_i ]_i just before [_{i-1})
   // We now know σ_{i+1}.
output w
```

(b) Decoding w given σ_w

```
Input: σ_w, m_1, ..., 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, ..., i
   receive_2(m_j)
   receive_1(m_j)
read_2 = ... [_{i-1} ]_{i-1} ...
ins_2(x between [_{i-1} and ]_{i-1})
send_2(m_x)
receive_2(m_x)
receive_1(m_x)
if read_1 = x∗
   return 1
else // read_1 = ∗x
   return 0
```

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

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

The proof of this result is nontrivial. It relies on an *inductive* coding argument, in which a string w is encoded in an execution. Later, the string w is recovered (decoded) bit by bit, using a feedback loop between two replica: one performs a black-box experiment on the protocol to recover the next bit of w, and the other reconstructs the corresponding steps the execution. The messages sent in the reconstructed part of the execution then form the basis decoding the next bit of w.

Theorem 8. *Let $\mathcal{R}$ be a push-based protocol with $n \geq 3$ replicas that satisfies the weak list specification. The worst-case metadata overhead of $\mathcal{R}$ over executions with D deletions is $\Omega(D)$.*

This follows from the following theorem.

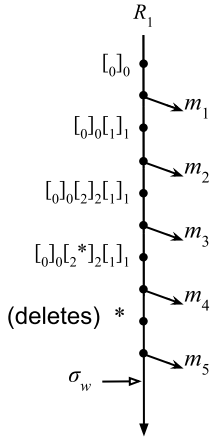
Theorem 9. *Let $\mathcal{R}$ be a push-based protocol with $n \geq 3$ replicas that satisfies the weak list specification. For every integer $D \geq 4$, and for every replica $R_1 \in \mathcal{R}$, there exists an execution α_D of $\mathcal{R}$ with D deletions such that: (1) the metadata overhead of some state σ of R_1 in α_D is $\Omega(D)$; (2) α_D satisfies causal atomic broadcast; and (3) R_1 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 $\Omega(D)$.

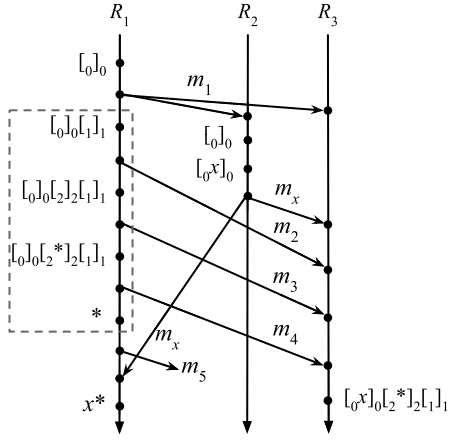
We show the existence of this execution using an information-theoretic argument. Namely, for every d-bit string w we construct an execution α_w that satisfies causal atomic broadcast and in which: (1) replica R_1 performs D deletions and receives no messages; (2) at the end of α_w, the user-observable 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. The metadata overhead of σ_{w_0} is therefore $\Omega(D)$, and thus α_{w_0} is the desired execution.

Encoding w. Given a d-bit string $w = w_1 \ldots w_d$, 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). Fig. 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. Fig. 5(a) depicts α_w for $w = 10$.

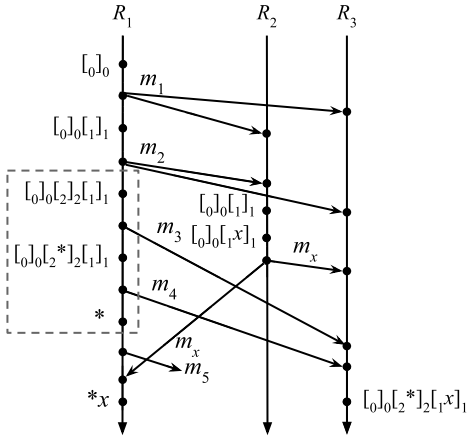
Only replica R_1 participates in the encoding execution α_w. We start by inserting the string $[_0\,]_0$ (i.e., the root). Because $\mathcal{R}$ is a push-based protocol, R_1 has a message m_1 pending following these insertions. We then proceed with a series of



(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.)

Fig. 5. Examples of the encoding and decoding procedures from Theorem 9 applied to $w = 10$. Fig. 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. Fig. 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 $w_1 = 1$. Fig. 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.

steps, for $i = 1, \ldots, d$. 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 $i = 0, \ldots, d$, 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 satisfies causal atomic broadcast by construction.

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 $m_1, \ldots, m_i$. (We describe Recover() in the next paragraph;

for now, assume it is an oracle.) Knowing w_i, in turn, determines the next event of R_1 in 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. Fig. 4(b) shows the pseudocode which, as it executes, decodes w. We start with R_1 in its initial state and reconstruct 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. Fig. 4(c) shows these state transitions, and Figs. 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^i_w; we will show that $[_{i-1}\,]_{i-1} \in v^i_w$. 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^i_w. We will show that y^i_w 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 Fig. 4(c) occur in an extension β^i_w of α_w of the form:

$$\begin{aligned}\beta^i_w = \alpha_w\, &receive_2(m_1)\, receive_1(m_1) \ldots receive_2(m_i)\, receive_1(m_i)\\ &do_2(\mathsf{read}_2, v^i_w)\, do_2(\mathsf{ins}_2(x, k^i_w), \_)\, send_2(m_x)\\ &receive_2(m_x)\, receive_1(m_x)\, do_1(\mathsf{read}_1, y^i_w),\end{aligned}$$

where k^i_w is the position at which R_2 inserts x into the list. In the following, we prove that the execution β^i_w is well-formed (Claim 10), that it satisfies causal atomic broadcast (Claim 11), that $[_{i-1}\,]_{i-1} \in v^i_w$ (Claim 12), and that $y^i_w = x*$ or $y^i_w = *x$ (Claim 13). 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^i_w, without having access to the entire execution α_w that leads R_1 to state σ_w.

Claim 10. *Execution β^i_w 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. □

Claim 11. *Execution β^i_w satisfies causal atomic broadcast.*

Proof. Immediate from inspection of the message delivery order in β^i_w. □

Claim 12. $[_{i-1}\,]_{i-1} \in v^i_w$.

Proof. For $j = 1, \ldots, d+1$, let $e_j, f_j \in \alpha_w$ be the *do* events in which R_1 inserts $[_{j-1}$ and $]_{j-1}$ into the list. Let $r \in \beta^i_w$ be the *do* event at which R_2 reads v^i_w. Because $\mathcal{R}$ is a correct push-based protocol and β^i_w satisfies causal atomic broadcast, by Definition 13, β^i_w 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^i_w, R_2 receives only the messages $m_1, \ldots, m_i$ before r, and each m_j is the first message sent by R_1 after e_j and f_j. It thus follows from the definition of the weak list specification (Definition 8) that $v^i_w = \ldots\, [_{i-1}\,]_{i-1} \ldots$. □

Claim 13. *$y^i_w = x*$ or $y^i_w = *x$.*

Proof. Let $f \in \beta^i_w$ be the *do* event at which R_2 inserts x into the list, and $r \in \beta^i_w$ be the *do* event at which R_1 reads y^i_w. Because $\mathcal{R}$ 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. □

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):

$$\gamma^i_w = \beta^i_w\, receive_3(m_1)\ \ldots\ receive_3(m_i)\, receive_3(m_x)$$
$$receive_3(m_{i+1})\ \ldots\ receive_3(m_{d+2})\, do_3(\mathsf{read}_3, z^i_w).$$

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 13); 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* model, replicas communicate only with a central server and not directly with each other. (The motivation is to maintain state on the server instead of on the replicas, and so the server usually does more than merely relay messages between replicas [20].) To model such protocols in our framework, which assumes a broadcast transport, we require replicas to process only messages to/from the server:

Definition 14. A protocol $\mathcal{R} = \{R_1, \ldots, R_n, S\}$ is a *client/server protocol* if for every replica $R_i = (Q^i, M, \Sigma^i, \sigma^i_0, E, \Delta^i)$, and $\sigma \in \Sigma^i$, if $\Delta^i(\sigma, receive(m)) \neq \sigma$, then m was sent by S. We call S the *server* and $R_1, \ldots, R_n$ the *clients*.

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 causal atomic broadcast, a broadcast protocol can *simulate* a client/server protocol using state machine replication [16].

Proposition 14. *Let $\mathcal{R} = \{R_1, \ldots, R_n, S\}$ be a client/server protocol. Then there exists a protocol $\mathcal{R}' = \{R'_1, \ldots, R'_n\}$ that* simulates *$\mathcal{R}$ 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 the respective sets of internal states of R_i and S; and (3) until R'_i receives a message, its state is $(\_, q^S_0)$, where $(q^S_0, \bot)$ is the initial state of S.*

Proof. For $i = 1 \ldots n$, 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.

Fig. 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. □

Client/server lower bound. Since the executions constructed in the proof of Theorem 9 satisfy causal atomic broadcast, they can also be viewed as executions of a protocol simulating a push-based client/server protocol (Proposition 14). We therefore obtain

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

Fig. 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.

Corollary 15. *Let $\mathcal{R}$ be a push-based client/server protocol with $n \geq 3$ replicas that satisfies the weak list specification. 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 list specification. Let $\mathcal{R}'$ be the protocol simulating $\mathcal{R}$ from Proposition 14. Take $D \geq 4$. By Theorem 9, 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 14 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. By definition of $\mathcal{R}'$, $do(\mathsf{read}, w)$ is enabled in σ if and only if it is enabled in q^R—that is, the user-observable content at R' and R is the same. It follows that the metadata overhead at R is $\Omega(D)$. □

7. Protocols with invisible reads in the presence of disconnections

Our metadata overhead lower bound proof applies to push-based protocols. However, being push-based is a low-level property, which partially specifies implementation details of the protocol. In this section, we show that under a weaker network model (discussed below), for a common class of protocols being push-based follows from guaranteeing the higher-level property of eventual visibility. Hence, our lower bound also applies to protocols satisfying the latter property.

We consider protocols with *invisible reads*, in which a read operation does not modify the state of the replica. We forbid both changing the replica's internal state and returning a response different from the response of the preceding user operation, unless a message was received in the meantime.

Definition 15. A protocol $\mathcal{R}$ has *invisible reads* if for every replica $R = (Q, M, \Sigma, \sigma_0, E, \Delta)$, if $do(\mathsf{read}, v)$ is enabled in state σ, then $\Delta(\sigma, do(\mathsf{read}, v)) = \sigma$, and further, if $\sigma = \Delta(\sigma', do(op, v'))$ for some $op \in \mathsf{Op}$, then $v = v'$.

Instead of a sufficiently connected network (Definition 1), we consider a network which guarantees the delivery of messages only to replicas that are *active*, as per the following definition; all other replicas are *disconnected*.

Definition 16. Let α be a well-formed execution of a protocol $\mathcal{R}$. We say that replica $R \in \mathcal{R}$ is *active* in α if R performs *do* events infinitely often in α.

Definition 17 *(Network model with disconnections).* The network is *sufficiently connected* in a well-formed execution α of a protocol $\mathcal{R}$ if the following conditions hold for every replica $R \in \mathcal{R}$: *(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 active replica $R' \neq R$ eventually receives m.

We adjust our definition of liveness (Definition 10) to the new network model.

Definition 18. A protocol $\mathcal{R}$ satisfying the weak list specification *guarantees eventual visibility under the network model with disconnections* if every execution α of $\mathcal{R}$ in which the network is sufficiently connected and that satisfies causal broadcast complies with some abstract execution $A \in \mathcal{A}_{\mathsf{weak}}$ that satisfies eventual visibility.

In the remainder of this section, we prove the following result.

Theorem 16. *Let $\mathcal{R}$ be a protocol with $n \geq 2$ replicas that has invisible reads, satisfies the weak list specification, and guarantees eventual visibility under the network model with disconnections. Then $\mathcal{R}$ is a push-based protocol.*

From this and Theorem 8, it follows that the metadata overhead bound holds under the conditions of Theorem 16.

Corollary 17. *Let $\mathcal{R}$ be a protocol with $n \geq 3$ replicas that has invisible reads, satisfies the weak list specification, and guarantees eventual visibility under the network model with disconnections. Then the worst-case metadata overhead of $\mathcal{R}$ over executions with D deletions is $\Omega(D)$.*

To prove Theorem 16, let $\mathcal{R}$ be a protocol with $n \geq 2$ replicas that has invisible reads, satisfies the weak list specification, and guarantees eventual visibility under the network model with disconnections. The following Lemmas 20 and 21 respectively establish Conditions 1, 2 and 3 of Definition 13, thereby proving that $\mathcal{R}$ is push-based. We start by stating some basic properties we rely on in our proofs.

Proposition 18 *([5, Proposition 1]). Let α be a well-formed execution of $\mathcal{R}$, and let e be an event in α. Consider β, the subsequence of α consisting of e and all events e' such that $e' \xrightarrow{\mathsf{hb}} e$. Then β is a well-formed execution of $\mathcal{R}$, and for any replica R, $\beta|_R$ is a prefix of $\alpha|_R$.*

Proposition 19. *Let α be an execution of $\mathcal{R}$, and let f be a $do(op, v)$ event such that $a \in v$. Then there exists an event $e = do(\mathsf{ins}(a, \_), \_) \in \alpha$ such that $e \xrightarrow{\mathsf{hb}} f$.*

Proof. The existence of some $e = do(\mathsf{ins}(a, \_), \_) \in \alpha$ is immediate: if no such e exists, any abstract execution that α complies with would not be in the weak list specification, which contradicts $\mathcal{R}$ satisfying the weak list specification. It remains to show that $e \xrightarrow{\mathsf{hb}} f$. Suppose that this is false. We show that $\mathcal{R}$ does not satisfy the weak list specification, which is a contradiction, as follows. Let α' be the subsequence of α consisting of f and all events e' such that $e' \xrightarrow{\mathsf{hb}} f$. By Proposition 18, α' is a well-formed execution. However, because inserted elements are unique, there is no insert of a in α'. Yet $a \in v$, where $f = do(op, v) \in \alpha'$, and so any abstract execution that α' complies with cannot be in the weak list specification. Then $\mathcal{R}$ does not satisfy the weak list specification, and the claim follows. □

Lemma 20. *Let α be an execution of $\mathcal{R}$ that satisfies causal broadcast, and let $e \in \alpha$. If one of the following holds, then $\mathsf{repl}(e)$ has a message pending after e: (1) $\mathsf{op}(e) = \mathsf{ins}(a, \_)$, or (2) $\mathsf{op}(e) = \mathsf{del}(a)$ and there does not exist $d \xrightarrow{\mathsf{hb}} e$ with $\mathsf{op}(d) = \mathsf{del}(a)$.*

Proof. Suppose that the claim is false. Then there exists $e \in \alpha$ such that (1) $\mathsf{op}(e) = \mathsf{ins}(a, \_)$, or (2) $\mathsf{op}(e) = \mathsf{del}(a)$ and there does not exist $d \xrightarrow{\mathsf{hb}} e$ with $\mathsf{op}(d) = \mathsf{del}(a)$, and $R = \mathsf{repl}(e)$ does not have a message pending after e. Let σ be the state of R after e (in which R does not have a message pending). Let α_1 be the subsequence of α consisting of e and all events e' such that $e' \xrightarrow{\mathsf{hb}} e$. By Proposition 18, α_1 is a well-formed execution, and $\alpha_1|_R$ is a prefix of $\alpha|_R$. Thus, R's state at the end of α_1 is also σ.

Consider some replica $R' \neq R$. Let $\beta = \alpha_1 \alpha_2$ be an execution of $\mathcal{R}$ that satisfies causal broadcast, obtained by appending *receive* events at R' to α_1 in some order that respects $\xrightarrow{\mathsf{hb}}$, so that in β, R' receives every message sent by another replica in α_1. Observe that because α satisfies causal broadcast, so does α_1, and so any message sent in α_1 is received by R before e. Thus, R does not receive messages in α_2 and so its state at the end of β remains σ. Thus, R does not have a message pending at the end of β.

Consider the infinite execution of $\mathcal{R}$, $\beta_\infty = \beta\, s\, r_1\, r_2 \ldots$, where the r_i events are reads at R', and, if R' has a message m pending after β, s is a *send* of m at R'; if R' does not have a message pending after β, $s = \varepsilon$. Because only R' performs infinitely many *do* events in β_∞, the network in β_∞ is sufficiently connected. 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 for e:

(1) $\mathsf{op}(e) = \mathsf{ins}(a, \_)$. Since R's state remains unchanged after e, it does not have a message pending and thus does not send a message after e. Therefore, $\neg(e \xrightarrow{\mathsf{hb}} r_j)$. Then by Proposition 19, $a \notin \mathsf{rval}(r_j)$. It follows from the weak list specification that there exists a $\mathsf{del}(a)$ event, $d \in H$, such that $d \xrightarrow{\mathsf{vis}} r_j$. Then $d \in \alpha_1$, since the suffix of β_∞ after α_1 contains only message transmissions, receipts, and read events. 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 $e' \xrightarrow{\mathsf{hb}} d$. Thus, $e' \in \alpha_1$. By Proposition 19, $e \xrightarrow{\mathsf{hb}} e'$. But this is a contradiction, since $e' \in \alpha_1$ implies $e' \xrightarrow{\mathsf{hb}} e$.

(2) $\mathsf{op}(e) = \mathsf{del}(a)$ *and there does not exist* $d \xrightarrow{\mathsf{hb}} e$ *with* $\mathsf{op}(d) = \mathsf{del}(a)$. 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 $g = \mathsf{ins}(a, \_) \in H$ such that $g \xrightarrow{\mathsf{vis}} e$, and so $g \xrightarrow{\mathsf{vis}} e \xrightarrow{\mathsf{vis}} r_j$. 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 in β_∞ there are no events at R after e. Now, let $A' = (H', \mathsf{vis}')$ be an abstract execution satisfying eventual visibility that β'_∞ complies with. By eventual visibility, g is visible to all but finitely many events. Therefore, there exists some r_i, $i \geq j$ such that $g \xrightarrow{\mathsf{vis}'} r_i$. Since $a \notin \mathsf{rval}(r_i)$, but $g \xrightarrow{\mathsf{vis}'} r_i$, it follows from the weak list specification that there exists a $\mathsf{del}(a)$ event, $d \in H$, such that $g \xrightarrow{\mathsf{vis}'} d \xrightarrow{\mathsf{vis}'} r_i$. Thus, $d \in \alpha_1$ and so $d \xrightarrow{\mathsf{hb}} e$, which contradicts our assumption about e. □

Lemma 21. *Let α be an execution of $\mathcal{R}$ that satisfies causal broadcast. Then there exists an abstract execution $A = (H, \mathsf{vis}) \in \mathcal{A}_{\mathsf{weak}}$ 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 $A \in \mathcal{A}_{\mathsf{weak}}$, we must show that it satisfies all conditions in Definition 8. We show that A satisfies Conditions b and c in Definition 7, as well as Condition 2 in Definition 8, by observing that these conditions depend only on the operations and their responses, and not on the visibility relation. Formally, because $\mathcal{R}$ satisfies the weak list specification, there exists an abstract execution $A' = (H', \mathsf{vis}') \in \mathcal{A}_{\mathsf{weak}}$ that α complies with. By definition of the weak specification, there exists a list order relation lo that satisfies Conditions b and c in Definition 7, as well as Condition 2 in Definition 8 for A'. Observe that these conditions only relate lo and the responses of the operations in H'; in particular, they do not depend on vis'. But because α complies with both A and A', for every replica R, $H|_R = H'|_R$. It thus follows that lo satisfies these conditions for A as well.

It remains to show that A satisfies Condition a. Abusing notation, we denote by $\mathsf{elems}(e)$ the set of elements in the response of a *do* event e. We thus need to show that for every *do* event $e \in \alpha$, $\mathsf{elems}(e) = L(e, \mathsf{vis})$, 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 a *do* event in α, let $R = \mathsf{repl}(e)$, and let σ be the state of R when the e transition occurs. Let α' be the subsequence of α consisting of e and all events e' such that $e' \xrightarrow{\mathsf{hb}} e$. By Proposition 18, α' is a well-formed execution of $\mathcal{R}$.

Consider now the infinite execution of $\mathcal{R}$, $\alpha_\infty = \alpha' s\, r_1\, r_2 \ldots$, where the r_i events are reads at R, and, if R has a message m pending after e, s is a *send* of m at R; if R does not have a message pending after e, $s = \varepsilon$. The network in α_∞ is sufficiently connected, because (1) only R performs infinitely many *do* events in α_∞, and (2) any message sent by some replica $R' \neq R$ in α' is received by R in α', since α satisfying causal broadcast implies that α' also does. Let $A' = (H', \mathsf{vis}') \in \mathcal{A}_{\mathsf{weak}}$ be an abstract execution satisfying eventual visibility that α_∞ complies with. Then there exists some r_j such that for every $e' \in \alpha'$, $e' \xrightarrow{\mathsf{vis}'} r_j$. Then $\mathsf{elems}(r_j) = L(r_j, \mathsf{vis}')$, by definition of the weak list specification.

Because $\mathcal{R}$ has invisible reads and the *send* in s', if it exists, does not change R's internal state, $\mathsf{elems}(r_j) = \mathsf{elems}(e)$. This implies that $\mathsf{elems}(e) = L(e, \mathsf{vis})$, because for every $e' \in \alpha'$, $e' \xrightarrow{\mathsf{vis}'} r_j \iff e' \leq_{\mathsf{hb}} e \iff e' \leq_{\mathsf{vis}} e$. □

8. Related work

Previous attempts at specifying the behavior of replicated list objects [30,18] 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. [6] 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 [5]) 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.

A mechanized proof of the eventual consistency property in RGA was presented in [13], following the initial publication of our work. There are other protocols implementing a highly available replicated list than the RGA protocol we considered. Treedoc [23] and Logoot [39] are other implementations of the strong list specification using the approach of replicated data types [28]. 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 the 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 position relative to an existing node). Like RGA, these protocols have the worst-case metadata overhead linear in the number of deletions. In particular, Logoot [39] can exhibit metadata overhead linear in the number of deletions despite not using tombstones for deletions. Instead, Logoot assigns elements with unique identifiers, and the size of these identifiers (which are metadata) can grow linearly with the number of deletions. WOOT [22] 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. Applying the transformation function may allow to commute two operations, a property called TP1, or three operations, a property called TP2. Though the idea of OT is simple, OT-based protocols are subtle and significant effort was done to gain a better understanding why some OT-based protocols work in centralized client/server situations [34,40]. Client/server OT protocols, such as Jupiter [20], allow the list state to converge if the transformation functions satisfy the TP1 property. Following the initial publication of our work, our formalism was used to show that Jupiter satisfies our weak list specification [38].

In peer-to-peer systems, OT protocols are more complicated. The dOPT protocol [12] does not work in all cases [25,29]. Indeed, it is impossible to design OT-based protocols for some peer-to-peer systems [24]. In peer-to-peer OT protocols, TP1 and TP2 are needed to ensure that the list state converges [25], regardless of the order in which the operations are received. Several OT protocols do not satisfy TP1 and TP2 and do not converge [15].

Sun et al. carry an in-depth study of OT- and CRDT-based collaborative editing systems, comparing the underlying concepts [33], complexity [32], and adoption in practice [31]. With respect to metadata overhead, they find that in the *common case*, the use of tombstones in CRDT-based systems leads to higher overhead than in OT-based systems [32]. Their observations are complementary to our work, which studies the fundamental *worst case* behavior of the system. In particular, although OT protocols do not use tombstones, they store a log of unacknowledged updates in each replica. Our proof (Section 6) essentially shows that in some worst-case executions, the overhead of these buffered updates can be linear in the number of deleted elements.

Our work considers the metadata overhead of collaborative editing systems. These systems have also been analyzed from the perspective of other complexity metrics, such as the time complexity of local and remote operations [2,8,14]. In some systems, however, the metadata overhead of the system influences the time complexity of its operations, e.g., when the time complexity depends on the size of object identifiers, which are part of the metadata [3].

9. 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 the RGA protocol [26]—as well as a *weak* list, which is implemented by the Jupiter protocol [20,38], underlying public collaboration systems [37].

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 to both weak and strong semantics. It is thus worth further investigating protocols for the client/server setting: even though systems such as Jupiter [20] do not provide strong semantics, our results suggest that this may 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.

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 [19]. While our lower bound would hold for more general systems, it is possible that the additional features induce additional complexity.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

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 ERC grant RACCOON. Yang was supported by an Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP, No. R0190-15-2011). Zawirski was supported by an EU project SyncFree.

Appendix A. Weak behavior in the Jupiter protocol

In this section, we give a concrete example of how the Jupiter protocol [20], which relies on OT, can produce the execution shown in Fig. 1(b). Jupiter is a client/server protocol, in which the client replicas and the server are connected via FIFO links. Each client operation is applied immediately locally and then sent to the server. The server totally orders received client operations and propagates the updates to the rest of replicas. To make sure all clients converge to the same state, Jupiter employs OT. Specifically, each participant (server or client) *transforms* incoming operations into "fixed up" operations that are applicable to its state [20,38]. While fully explaining Jupiter's transformations is beyond the scope of this paper, we illustrate the idea in the following example, which shows how the execution of Fig. 1(b) can occur in Jupiter.

Initially, all clients and servers have an empty list. Client R_2 puts x into the list and this operation is received by the server, which propagates it to the other clients, R_1 and R_3:

$$R_2.do(\mathsf{ins}(x, 0))$$
$$S.receive(R_2, \epsilon, \mathsf{ins}(x, 0))$$
$$R_1.receive(S, \epsilon, \mathsf{ins}(x, 0))$$
$$R_3.receive(S, \epsilon, \mathsf{ins}(x, 0))$$

A *receive* event at replica R consists of a tuple (X, σ, o), where X is the sending replica, σ is a list state, and o is an operation. The state σ reflects the state of the list constructed from all updates that X has received from R at the time of sending its message, and allows R to transform X's update appropriately. We show update messages as containing list states for readability. In practice, messages contain an acknowledgment number indicating which of R's updates X has received, and R buffers its sent messages until they are acknowledged, allowing it to reconstruct list states from received acknowledgments.

Continuing the example, R_2 now deletes x and sends an update to the server. An important detail is that, in contrast to the idealized depiction in Fig. 1, Jupiter identifies deletions with the index of the character deleted, and not by the deleted character.

$$R_2.do(\mathsf{del}(0))$$
$$S.receive(R_2, x, \mathsf{del}(0))$$

Above, we show R_2's update as reflecting an x state even though R_2 did not receive an update from the server about the insertion of x. The reason is that x was inserted into the list by R_2, which the server is aware of.

Following the above events, the list state at the server is ϵ. Next, R_1 inserts a to the left of x and R_3 inserts b to the right of x, and both send updates to the server:

$$R_1.do(\mathsf{ins}(a, 0))$$
$$R_3.do(\mathsf{ins}(b, 1))$$

Next, the server receives R_1's update before receiving R_3's update:

$$S.receive(R_1, x, \mathsf{ins}(a, 0))$$
$$S.receive(R_3, x, \mathsf{ins}(b, 1))$$

Upon receiving R_1's update, the server's list state is updated to a. Crucially, when R_3's update is received, it is *transformed*: since b was inserted at index when the list was x and x was since deleted, the server decrements the index of R_3's ins operation, and applies the operation $\mathsf{ins}(b, 0)$ to its state. The result is that the server's list state is now ba. The server propagates the updates to the other replicas, which perform similar transformations, resulting in all replicas converging on the list state ba.

References

[1] Apache Wave, https://incubator.apache.org/wave/.
[2] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, P. Urso, Evaluating CRDTs for real-time document editing, in: Proceedings of the 11th ACM Symposium on Document Engineering, DocEng 2011, 2011.
[3] L. André, S. Martin, G. Oster, C.-L. Ignat, Supporting adaptable granularity of changes for massive-scale collaborative editing, in: Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom 2013, IEEE, 2013.
[4] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, M. Zawirski, Specification and complexity of collaborative text editing, in: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, 2016, pp. 259–268.
[5] H. Attiya, F. Ellen, A. Morrison, Limitations of highly-available eventually-consistent data stores, IEEE Trans. Parallel Distrib. Syst. 28 (1) (2017) 141–155.
[6] S. Burckhardt, A. Gotsman, H. Yang, M. Zawirski, Replicated data types: specification, verification, optimality, in: POPL, 2014.
[7] C. Cachin, R. Guerraoui, L. Rodrigues, Introduction to Reliable and Secure Distributed Programming, 2nd edition, Springer Publishing Company, Incorporated, 2011.
[8] Q.-V. Dang, C.-L. Ignat, Performance of real-time collaborative editors at large scale: user perspective, in: 2016 IFIP Networking Conference (IFIP Networking) and Workshops, 2016.
[9] 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.
[10] 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.
[11] X. Défago, A. Schiper, P. Urbán, Total order broadcast and multicast algorithms: taxonomy and survey, ACM Comput. Surv. 36 (4) (2004).
[12] C.A. Ellis, S.J. Gibbs, Concurrency control in groupware systems, in: SIGMOD, 1989.
[13] V.B. Gomes, M. Kleppmann, D.P. Mulligan, A.R. Beresford, Verifying strong eventual consistency in distributed systems, in: OOPSLA, 2017.
[14] C.-L. Ignat, G. Oster, O. Fox, V.L. Shalin, F. Charoy, How do user groups cope with delay in real-time collaborative note taking, in: Proceedings of the 14th European Conference on Computer Supported Cooperative Work, ECSCW 2015, 2015.
[15] A. Imine, M. Rusinowitch, G. Oster, P. Molli, Formal design and verification of operational transformation algorithms for copies convergence, Theor. Comput. Sci. 351 (2) (2006).

[16] L. Lamport. Time, Clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978).
[17] B. Leuf, W. Cunningham, The Wiki Way: Quick Collaboration on the Web, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
[18] D. Li, R. Li, Preserving operation effects relation in group editors, in: CSCW, 2004.
[19] S. Martin, P. Urso, S. Weiss, Scalable XML collaborative editing with undo, in: OTM, 2010.
[20] D.A. Nichols, P. Curtis, M. Dixon, J. Lamping, High-latency, low-bandwidth windowing in the Jupiter collaboration system, in: UIST, 1995.
[21] G. Oster, P. Molli, P. Urso, A. Imine, Tombstone transformation functions for ensuring consistency in collaborative editing systems, in: Proceedings of the 2nd IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom 2006, IEEE, 2006, pp. 1–10.
[22] G. Oster, P. Urso, P. Molli, A. Imine, Data consistency for P2P collaborative editing, in: CSCW, 2006.
[23] N. Preguiça, J.M. Marqués, M. Shapiro, M. Leția, A commutative replicated data type for cooperative editing, in: ICDCS, 2009.
[24] A. Randolph, H. Boucheneb, A. Imine, Q. Alejandro, On consistency of operational transformation approach, in: INFINITY 2012-14th International Workshop on Verification of Infinite-State Systems, 2012, pp. 1–15.
[25] M. Ressel, D. Nitsche-Ruhland, R. Gunzenhäuser, An integrating, transformation-oriented approach to concurrency control and undo in group editors, in: CSCW, 1996.
[26] H.-G. Roh, M. Jeon, J.-S. Kim, J. Lee, Replicated abstract data types: building blocks for collaborative applications, J. Parallel Distrib. Comput. 71 (3) (2011).
[27] B. Shao, D. Li, T. Lu, N. Gu, An operational transformation based synchronization protocol for web 2.0 applications, in: Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work, CSCW 2011, 2011.
[28] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, Conflict-free replicated data types, in: SSS, 2011.
[29] C. Sun, C. Ellis, Operational transformation in real-time group editors: issues, algorithms, and achievements, in: CSCW, 1998.
[30] C. Sun, X. Jia, Y. Zhang, Y. Yang, D. Chen, Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems, ACM Trans. Comput.-Hum. Interact. 5 (1) (1998).
[31] C. Sun, D. Sun Agustina, W. Cai, Real differences between OT and CRDT in building co-editing systems and real world applications, CoRR, arXiv: 1905.01517 [abs], 2019.
[32] C. Sun, D. Sun Agustina, W. Cai, Real differences between OT and CRDT in correctness and complexity for consistency maintenance in co-editors, CoRR, arXiv:1905.01302 [abs], 2019.
[33] C. Sun, D. Sun Agustina, W. Cai, Real differences between OT and CRDT under a general transformation framework for consistency maintenance in co-editors, CoRR, arXiv:1905.01518 [abs], 2019.
[34] D. Sun, C. Sun, Context-based operational transformation in distributed collaborative editing systems, IEEE Trans. Parallel Distrib. Syst. 20 (10) (2008) 1454–1470.
[35] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, C.H. Hauser, Managing update conflicts in Bayou, a weakly connected replicated storage system, in: SOSP, 1995.
[36] W. Vogels, Eventually consistent, Commun. ACM 52 (1) (2009).
[37] D. Wang, A. Mah, S. Lassen, Google wave operational transformation, https://wave-protocol.googlecode.com/hg/whitepapers/operational-transform/operational-transform.html, 2010.
[38] H. Wei, Y. Huang, J. Lu, Specification and implementation of replicated list: the Jupiter protocol revisited, in: OPODIS, 2018.
[39] S. Weiss, P. Urso, P. Molli Logoot, A scalable optimistic replication algorithm for collaborative editing on P2P networks, in: ICDCS, 2009.
[40] Y. Xu, C. Sun, M. Li, Achieving convergence in operational transformation: conditions, mechanisms and systems, in: Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing, CSCW '14, 2014, pp. 505–518.