# RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks*

Nancy Lynch[1] and Alex A. Shvartsman[2,1]

[1] Laboratory for Computer Science, Massachusetts Institute of Technology, 200 Technology Square, NE43-365, Cambridge, MA 02139, USA.
[2] Department of Computer Science and Engineering, University of Connecticut, 191 Auditorium Road, Unit 3155, Storrs, CT 06269, USA.

**Abstract.** This paper presents an algorithm that emulates atomic read/write shared objects in a dynamic network setting. To ensure availability and fault-tolerance, the objects are replicated. To ensure atomicity, reads and writes are performed using *quorum configurations*, each of which consists of a set of *members* plus sets of *read-quorums* and *write-quorums*. The algorithm is *reconfigurable*: the quorum configurations may change during computation, and such changes do not cause violations of atomicity. Any quorum configuration may be installed at any time. The algorithm tolerates processor stopping failure and message loss. The algorithm performs three major tasks, all concurrently: reading and writing objects, introducing new configurations, and "garbage-collecting" obsolete configurations. The algorithm guarantees atomicity for arbitrary patterns of asynchrony and failure. The algorithm satisfies a variety of conditional performance properties, based on timing and failure assumptions. In the "normal case", the latency of read and write operations is at most $8d$, where $d$ is the maximum message delay.

## 1 Introduction

This paper presents an algorithm that can be used to implement atomic read/write shared memory in a dynamic network setting, in which participants may join or fail during the course of computation. Examples of such settings are mobile networks and peer-to-peer networks. One use of this service might be to provide long-lived data in a dynamic and volatile setting such as a military operation.

In order to achieve availability in the presence of failures, the objects are replicated. In order to maintain memory consistency in the presence of small and transient changes, the algorithm uses *configurations*, each of which consists of a set of *members* plus sets of *read-quorums* and *write-quorums*. In order to accommodate larger and more permanent changes, the algorithm supports *reconfiguration*, by which the set of members and the sets of quorums are modified. Such changes do not cause violations of atomicity. Any quorum configuration may be installed at any time.

* This work was supported by the NSF ITR Grant 0121277. The work of the first author was also supported by AFOSR under contract F49620-00-1-0097 and by NTT under contract MIT9904-12. The work of the second author was also supported by NSF CAREER Award 9984778 and NSF Grant 9988304.

D. Malkhi (Ed.): DISC 2002, LNCS 2508, pp. 173–190, 2002.

We first provide a formal specification for reconfigurable atomic shared memory as a global service. We call this service RAMBO, which stands for "Reconfigurable Atomic Memory for Basic Objects" ("Basic" means "Read/Write"). The rest of the paper presents our algorithm and its analysis. The algorithm carries out three major activities, all concurrently: reading and writing objects, introducing new configurations, and removing ("garbage-collecting") obsolete configurations. The algorithm is composed of a *main algorithm*, which handles reading, writing, and garbage-collection, and a global reconfiguration service, $Recon$, which provides the main algorithm with a consistent sequence of configurations. Reconfiguration is loosely coupled to the main algorithm, in particular, several configurations may be known to the algorithm at one time, and read and write operations can use them all.

The main algorithm performs read and write operations using a two-phase strategy. The first phase gathers information from read-quorums of active configurations and the second phase propagates information to write-quorums of active configurations. This communication is carried out using background gossiping, which allows the algorithm to maintain only a small amount of protocol state information. Each phase is terminated by a *fixed point* condition that involves a quorum from each active configuration. Different read and write operations may execute concurrently: the restricted semantics of reads and writes permit the effects of this concurrency to be sorted out afterwards.

The main algorithm also includes a facility for *garbage-collecting* old configurations when their use is no longer necessary for maintaining consistency. Garbage-collection also uses a two-phase strategy, where the first phase communicates with an old configuration and the second phase communicates with a new configuration. A garbage-collection operation ensures that both a read-quorum and a write-quorum of the old configuration learn about the new configuration, and that the latest value from the old configuration is conveyed to a write-quorum of the new configuration.

The reconfiguration service is implemented by a distributed algorithm that uses distributed consensus to agree on the successive configurations. Any member of the latest configuration $c$ may propose a new configuration at any time; different proposals are reconciled by an execution of consensus among the members of $c$. Consensus is, in turn, implemented using a version of the Paxos algorithm [17], as described formally in [8]. Although such consensus executions may be slow—in fact, in some situations, they may not even terminate—they do not delay read and write operations.

We show atomicity for arbitrary patterns of asynchrony and failure. We analyze performance *conditionally*, based on timing and failure assumptions. For example, assuming that gossip and garbage-collection occur periodically, that reconfiguration is requested infrequently enough for garbage-collection to keep up, and that quorums of active configurations do not fail, we show that read and write operations complete within time $8d$, where $d$ is the maximum message latency.

**Comparison with other approaches.** Consensus algorithms can be used directly to implement an atomic data service by allowing participants to agree on a global total ordering of all operations [17]. In contrast, we use consensus to agree only on the sequence of configurations and not on the individual operations. Also, in our algorithm, the termination of consensus affects the termination of reconfiguration, but not of read and write operations.

Group communication services (GCSs) [1] can also be used to implement an atomic data service, e.g., by implementing a global totally ordered broadcast service on top of a view-synchronous GCS [11] using techniques of [16,2]. In most GCS implementations, forming a new view takes a substantial amount of time, and client-level operations are delayed during the view-formation period. In our algorithm, reads and writes can make progress during reconfiguration. Also, in some standard GCS implementations, e.g., [5], performance is degraded even if only one stopping failure occurs; our algorithm uses quorums to tolerate small numbers of failures.

A dynamic primary configuration GCS was introduced in [7] and used to implement atomic memory, using techniques of [3] within each configuration. That work restricts the set of possible new configurations to those satisfying certain intersection properties with respect to the previous configurations, whereas we impose no such restrictions. Like other solutions based on GCSs, the algorithm of [7] delays reads and writes during reconfiguration.

*Single reconfigurer* implementations for atomic memory are considered in [19,10]. In these approaches, the failure of the reconfigurer disables future reconfiguration. Also, in [19,10], garbage-collection of an old configuration is tightly coupled to the introduction of a new configuration, whereas in our new algorithm, garbage-collection is carried out in the background, concurrently with other processing.

**Other related work.** The first general scheme for emulating shared memory in message-passing systems by using replication and accessing majorities of time-stamped replicas was given in [21]. An algorithm for majority-based emulation of atomic read/write memory was presented in [3]. This algorithm introduced a two-phase paradigm in which the first phase gathers information from a majority of participants, and the second phase propagates information to a majority. A *quorum system* [13]—a generalization of majority sets—is a collection of sets such that any two sets, called *quorums*, intersect [12]. Quorum systems have been used to implement data replication protocols, e.g., [4,6,14]. Consensus algorithms have been used as building blocks in other work, e.g, [15].

**Note.** This paper is an extended abstract of a full report [20]. The full version includes specifications of all components, complete proofs, and additional results.

## 2 Data Types

We assume distinguished elements $\perp$ and $\pm$, which are not in any of the basic types. For any type $A$, we define types $A_\perp = A \cup \{\perp\}$. and $A_\pm = A \cup \{\perp, \pm\}$. If $A$ is a poset, we augment its ordering by assuming that $\perp < a < \pm$ for every $a \in A$.

We assume the following data types and distinguished elements: $I$, the totally-ordered set of *locations*. $T$, the set of *tags*, defined as $\mathbb{N} \times I$. $M$, the set of *messages*. $X$, the set of *object identifiers*, partitioned into subsets $X_i$, $i \in I$; $X_i$ is the set of identifiers for objects that may be created at location $i$. For any $x \in X$, $(i_0)_x$ denotes the unique $i$ such that $x \in X_i$. For each $x \in X$, we define $V_x$, the set of values that object $x$ may take on, and $(v_0)_x \in V_x$, the initial value of $x$.

We also assume: $C$, the set of *configuration ids*; we use the trivial partial order on $C$, in which all elements are incomparable. For each $x \in X$, $(c_0)_x \in C$, the *initial configuration id* for $x$. For each $c \in C$: $members(c)$, a finite subset of $I$,

$read\text{-}quorums(c)$ and $write\text{-}quorums(c)$, two sets of finite subsets of $members(c)$. We assume: (1) $members((c_0)_x) = \{(i_0)_x\}$, that is, the initial configuration for $x$ has one member, the creator of $x$. (2) For every $c$, every $R \in read\text{-}quorums(c)$, and every $W \in write\text{-}quorums(c)$, $R \cap W \neq \emptyset$.

We define functions and sets for configurations: $update$, a binary function on $C_\pm$, defined by $update(c, c') = \max(c, c')$ if $c$ and $c'$ are comparable, $update(c, c') = c$ otherwise. $extend$, a binary function on $C_\pm$, defined by $extend(c, c') = c'$ if $c = \perp$ and $c' \in C$, and $extend(c, c') = c$ otherwise. $CMap$, the set of *configuration maps*, defined as $\mathbb{N} \to C_\pm$. We extend the $update$ and $extend$ operators elementwise to binary operations on $CMap$. $truncate$, a unary function on $CMap$, defined by $truncate(cm)(k) = \perp$ if there exists $\ell \leq k$ such that $cm(\ell) = \perp$, $truncate(cm)(k) = cm(k)$ otherwise. Truncation removes all the configuration ids that follow a $\perp$. $Truncated$, the subset of $CMap$ such that $cm \in Truncated$ if and only if $truncate(cm) = cm$. $Usable$, the subset of $CMap$ such that $cm \in Usable$ iff the pattern occurring in $cm$ consists of a prefix of finitely many $\pm$s, followed by an element of $C$, followed by an infinite sequence of elements of $C_\perp$ in which all but finitely many elements are $\perp$.

## 3 Reconfigurable Atomic Memory Service Specification

Our specification for the RAMBO service consists of an external signature plus a set of traces that embody RAMBO's safety properties. No liveness properties are included; we replace these with conditional latency bounds, which appear in Section 8. The external signature appears in Figure 1. We use I/O automata notation for all our specifications.

Input:
- $\mathsf{join}(\mathsf{rambo}, J)_{x,i}$, $J$ a finite subset of $I - \{i\}$, $x \in X$, $i \in I$, such that if $i = (i_0)_x$ then $J = \emptyset$
- $\mathsf{read}_{x,i}$, $x \in X, i \in I$
- $\mathsf{write}(v)_{x,i}$, $v \in V_x, x \in X, i \in I$
- $\mathsf{recon}(c, c')_{x,i}$, $c, c' \in C, i \in members(c), x \in X, i \in I$
- $\mathsf{fail}_i$, $i \in I$

Output:
- $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,i}, x \in X, i \in I$
- $\mathsf{read\text{-}ack}(v)_{x,i}, v \in V_x, x \in X, i \in I$
- $\mathsf{write\text{-}ack}_{x,i}, x \in X, i \in I$
- $\mathsf{recon\text{-}ack}(b)_{x,i}, b \in \{\mathsf{ok}, \mathsf{nok}\}, x \in X, i \in I$
- $\mathsf{report}(c)_{x,i}, c \in C, c \in X, i \in I$

**Fig. 1.** RAMBO$(x)$: External signature

The client at location $i$ requests to join the system for a particular object $x$ by performing a $\mathsf{join}(\mathsf{rambo}, J)_{x,i}$ input action. The set $J$ represents the client's guess at a set of processes that have already joined the system for $x$. If $i = (i_0)_x$, the set $J$ is empty, because $(i_0)_x$ is supposed to be the first process to join the system for $x$. If the join attempt is successful, the RAMBO service responds with a $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,i}$ output action. The client at $i$ initiates a read (resp., write) operation using a $\mathsf{read}_i$ (resp., $\mathsf{write}_i$) input action, which the RAMBO service acknowledges with a $\mathsf{read\text{-}ack}_i$ (resp., $\mathsf{write\text{-}ack}_i$) output. The client initiates a reconfiguration using a $\mathsf{recon}_i$ input, which is acknowledged with a $\mathsf{recon\text{-}ack}_i$ output. RAMBO reports a new configuration to the client using a $\mathsf{report}_i$ output. Finally, a stopping failure at location $i$ is modelled using a $\mathsf{fail}_i$ input action. We model process "leaves" as failures.

The set of traces describing RAMBO's safety properties consists of those that satisfy an implication of the form "environment assumptions imply service guarantees". The environment assumptions are simple "well-formedness" conditions:

*Well-formedness:* (1) For every $x$ and $i$: (a) No $\mathsf{join}(\mathsf{rambo}, *)_{x,i}$, $\mathsf{read}_{x,i}$, $\mathsf{write}(*)_{x,i}$, or $\mathsf{recon}(*,*)_{x,i}$ event is preceded by a $\mathsf{fail}_i$ event. (b) At most one $\mathsf{join}(\mathsf{rambo}, *)_{x,i}$ event occurs. (c) Any $\mathsf{read}_{x,i}$, $\mathsf{write}(*)_{x,i}$, or $\mathsf{recon}(*,*)_{x,i}$ event is preceded by a $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,i}$ event. (d) Any $\mathsf{read}_{x,i}$, $\mathsf{write}(*)_{x,i}$, or $\mathsf{recon}(*,*)_{x,i}$ event is preceded by an -ack event for any preceding event of any of these kinds. (2) For every $x$ and $c$, at most one $\mathsf{recon}(*,c)_{x,*}$ event occurs. (Configuration ids that are proposed in recon events are unique. This is not a serious restriction, because the same membership and quorum sets may be associated with different configuration ids.) (3) For every $c$, $c'$, $x$, and $i$, if a $\mathsf{recon}(c,c')_{x,i}$ event occurs, then it is preceded by a $\mathsf{report}(c)_{x,i}$ event and by a $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,j}$ event for every $j \in members(c')$.

The safety guarantees provided by the service are as follows:

*Well-formedness:* For every $x$ and $i$: (a) No $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,i}$, $\mathsf{read\text{-}ack}(*)_{x,i}$, $\mathsf{write\text{-}ack}_{x,i}$, $\mathsf{recon\text{-}ack}(*)_{x,i}$, or $\mathsf{report}(*)_{x,i}$ event is preceded by a $\mathsf{fail}_i$ event. (b) Any $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,i}$ (resp., $\mathsf{read\text{-}ack}(*)_{x,i}$, $\mathsf{write\text{-}ack}_{x,i}$, $\mathsf{recon\text{-}ack}(*)_{x,i}$) event has a preceding $\mathsf{join}(\mathsf{rambo}, *)_{x,i}$ (resp., $\mathsf{read}_{x,i}$, $\mathsf{write}(*)_{x,i}$, $\mathsf{recon}(*,*)_{x,i}$) event with no intervening invocation or response action for $x$ and $i$.

*Atomicity:*[1] If all the read and write operations that are invoked complete, then the read and write operations for object $x$ can be partially ordered by an ordering $\prec$, so that the following conditions are satisfied: (1) No operation has infinitely many other operations ordered before it. (2) The partial order is consistent with the external order of invocations and responses, that is, there do not exist read or write operations $\pi_1$ and $\pi_2$ such that $\pi_1$ completes before $\pi_2$ starts, yet $\pi_2 \prec \pi_1$. (3) All write operations are totally ordered and every read operation is ordered with respect to all the writes. (4) Every read operation ordered after any writes returns the value of the last write preceding it in the partial order; any read operation ordered before all writes returns $(v_0)_x$.

The rest of the paper presents our implementation of RAMBO. The implementation is a distributed algorithm in the asynchronous message-passing model. All processes may communicate with each other. Processes may fail by stopping without warning.

Our implementation can be described formally as the composition of a separate implementation for each $x$, so we describe the implementation for a generic $x$. We suppress explicit mention of $x$, writing $V$, $v_0$, $c_0$, and $i_0$ as shorthand for $V_x$, $(v_0)_x$, $(c_0)_x$, and $(i_0)_x$, respectively.

[1] Atomicity is often defined in terms of an equivalence with a serial memory. The definition given here implies this equivalence, as shown, for example, in Lemma 13.16 in [18]. Although Lemma 13.16 of [18] is presented for a setting with only finitely many locations, nothing in Lemma 13.16 or its proof depends on the finiteness of the set of locations.

## 4 Reconfiguration Service Specification

Our RAMBO implementation for object $x$ consists of a main $RW$ algorithm and a reconfiguration service, $Recon$. Here we present the specification for $Recon$. Our implementation of $Recon$ is described in Section 7.

The external signature appears in Figure 2. The client of $Recon$ at location $i$ requests to join the reconfiguration service by performing a $\mathsf{join}(\mathsf{recon})_i$ input action. The service acknowledges this with a corresponding $\mathsf{join\text{-}ack}_i$ output action. The client requests reconfiguration using a $\mathsf{recon}_i$ input, which is acknowledged with a $\mathsf{recon\text{-}ack}_i$ output action. The service reports a new configuration to the client using a $\mathsf{report}_i$ output action. Outputs of the form $\mathsf{new\text{-}config}(c,k)_i$ announce at location $i$ that $c$ is the $k^{th}$ configuration id. These outputs are used for communication with the portion of the $RW$ algorithm running at location $i$. $Recon$ announces consistent information, only one configuration id for each index in the configuration id sequence. $Recon$ delivers information about each configuration to members of the new configuration and members of the immediately preceding configuration. Crashes are modeled using fail actions.

Input:
- $\mathsf{join}(\mathsf{recon})_i, i \in I$
- $\mathsf{recon}(c,c')_i, c, c' \in C, i \in members(c)$
- $\mathsf{fail}_i, i \in I$

Output:
- $\mathsf{join\text{-}ack}(\mathsf{recon})_i, i \in I$
- $\mathsf{recon\text{-}ack}(b)_i, b \in \{\mathsf{ok}, \mathsf{nok}\}, i \in I$
- $\mathsf{report}(c)_i, c \in C, i \in I$
- $\mathsf{new\text{-}config}(c,k)_i, c \in C, k \in \mathbb{N}^+, i \in I$

**Fig. 2.** $Recon$: External signature

The set of traces describing $Recon$'s safety properties is defined by environment assumptions and service guarantees. The environment assumptions are simple well-formedness conditions, consistent with those for RAMBO:

*Well-formedness:* (1) For every $i$: (a) No $\mathsf{join}(\mathsf{recon})_i$ or $\mathsf{recon}(*,*)_i$ event is preceded by a $\mathsf{fail}_i$ event. (b) At most one $\mathsf{join}(\mathsf{recon})_i$ event occurs. (c) Any $\mathsf{recon}(*,*)_i$ event is preceded by a $\mathsf{join\text{-}ack}(\mathsf{recon})_i$ event. (d) Any $\mathsf{recon}(*,*)_i$ event is preceded by an -ack for any preceding $\mathsf{recon}(*,*)_i$ event. (2) For every $c$, at most one $\mathsf{recon}(*,c)_*$ event occurs. (3) For every $c$, $c'$, $x$, and $i$, if a $\mathsf{recon}(c,c')_i$ event occurs, then it is preceded by: (a) A $\mathsf{report}(c)_i$ event, and (b) A $\mathsf{join\text{-}ack}(\mathsf{recon})_j$ for every $j \in members(c')$.

The safety guarantees are:

*Well-formedness:* For every $i$: (a) No $\mathsf{join\text{-}ack}(\mathsf{recon})_i$, $\mathsf{recon\text{-}ack}(*)_i$, $\mathsf{report}(*)_i$, or $\mathsf{new\text{-}config}(*,*)_i$ event is preceded by a $\mathsf{fail}_i$ event. (b) Any $\mathsf{join\text{-}ack}(\mathsf{recon})_i$ (resp., $\mathsf{recon\text{-}ack}(c)_i$) event has a preceding $\mathsf{join}(\mathsf{recon})_i$ (resp., $\mathsf{recon}_i$) event with no intervening invocation or response action for $x$ and $i$.

*Agreement:* If $\mathsf{new\text{-}config}(c,k)_i$ and $\mathsf{new\text{-}config}(c',k)_j$ both occur, then $c = c'$.

*Validity:* If $\mathsf{new\text{-}config}(c,k)_i$ occurs, then it is preceded by a $\mathsf{recon}(*,c)_{i'}$ for some $i'$ for which a matching $\mathsf{recon\text{-}ack}(\mathsf{nok})_{i'}$ does not occur.

*No duplication:* If $\mathsf{new\text{-}config}(c,k)_i$ and $\mathsf{new\text{-}config}(c,k')_{i'}$ both occur, then $k = k'$.

# 5 Implementation of RAMBO Using a Reconfiguration Service

Our RAMBO implementation includes, for each $i$, a $Joiner_i$ automaton, which handles joining, and a $RW_i$ automaton, which handles reading, writing, and "installing" new configurations. These automata use asynchronous communication channels $Channel_{i,j}$. The $RW$ automata also interact with an arbitrary implementation of $Recon$.

## 5.1 Joiner Automata

When $Joiner_i$ receives a $\mathsf{join}(\mathsf{rambo}, J)$ request from its environment, it sends join messages to the processes in $J$ (with the hope that they are already participating, and so can help in its attempt to join). Also, it submits join requests to the local $RW$ and $Recon$ components and waits for acknowledgments. The join messages that are sent by $Joiner$ automata are handled by $RW$ automata at other locations.

## 5.2 Reader-Writer Automata

$RW_i$ processes each read or write operation using one or more configurations, which it learns about from the $Recon$ service. It also handles the garbage-collection of older configurations. The signature and state of $RW_i$ appear in Figure 3. Figure 4 presents the transitions pertaining to joining the protocol and failing. Figure 5 presents those pertaining to reading and writing, and Figure 6 presents those pertaining to garbage-collection.

**Signature:**

Input:
- $\mathsf{read}_i$
- $\mathsf{write}(v)_i, v \in V$
- $\mathsf{new\text{-}config}(c,k)_i, c \in C, k \in \mathbb{N}^+$
- $\mathsf{recv}(\mathsf{join})_{j,i}, j \in I - \{i\}$
- $\mathsf{recv}(m)_{j,i}, m \in M, j \in I$
- $\mathsf{join}(\mathsf{rw})_i$
- $\mathsf{fail}_i$

Output:
- $\mathsf{join\text{-}ack}(\mathsf{rw})_i$
- $\mathsf{read\text{-}ack}(v)_i, v \in V$
- $\mathsf{write\text{-}ack}_i$
- $\mathsf{send}(m)_{i,j}, m \in M, j \in I$

Internal:
- $\mathsf{query\text{-}fix}_i$
- $\mathsf{prop\text{-}fix}_i$
- $\mathsf{gc}(k)_i, k \in \mathbb{N}$
- $\mathsf{gc\text{-}query\text{-}fix}(k)_i, k \in \mathbb{N}$
- $\mathsf{gc\text{-}prop\text{-}fix}(k)_i, k \in \mathbb{N}$
- $\mathsf{gc\text{-}ack}(k)_i, k \in \mathbb{N}$

**State:**

- $status \in \{\mathsf{idle}, \mathsf{joining}, \mathsf{active}\}$, initially idle
- $world$, a finite subset of $I$, initially $\emptyset$
- $value \in V$, initially $v_0$
- $tag \in T$, initially $(0, i_0)$
- $cmap \in CMap$, initially
  - $cmap(0) = c_0$,
  - $cmap(k) = \perp$ for $k \geq 1$
- $pnum1 \in \mathbb{N}$, initially 0
- $pnum2 \in I \to \mathbb{N}$, initially everywhere 0
- $failed$, a Boolean, initially $false$

$op$, a record with fields:
- $type \in \{\mathsf{read}, \mathsf{write}\}$
- $phase \in \{\mathsf{idle}, \mathsf{query}, \mathsf{prop}, \mathsf{done}\}$, initially idle
- $pnum \in \mathbb{N}$
- $cmap \in CMap$
- $acc$, a finite subset of $I$
- $value \in V$

$gc$, a record with fields:
- $phase \in \{\mathsf{idle}, \mathsf{query}, \mathsf{prop}\}$, initially idle
- $pnum \in \mathbb{N}$
- $acc$, a finite subset of $I$
- $index \in \mathbb{N}$

**Fig. 3.** $RW_i$: Signature and state

**State variables.** The $status$ variable keeps track of the progress of the component as it joins the protocol. When $status = \mathsf{idle}$, $RW_i$ does not respond to any inputs (except

for join) and does not perform any locally controlled actions. When $status = \mathsf{joining}$, $RW_i$ responds to inputs but still does not perform any locally controlled actions. When $status = \mathsf{active}$, the automaton participates fully in the protocol.

The $world$ variable is used to keep track of all processes that are known to have tried to join the system. The $value$ variable contains the current value of the local replica of $x$, and $tag$ holds the associated tag.

The $cmap$ variable contains information about configurations. If $cmap(k) = \perp$, it means that $RW_i$ has not yet learned what the $k^{th}$ configuration id is. If $cmap(k) = c \in C$, it means that $RW_i$ has learned that the $k^{th}$ configuration id is $c$, and has not yet garbage-collected it. If $cmap(k) = \pm$, it means that $RW_i$ has garbage-collected the $k^{th}$ configuration id. $RW_i$ learns about configuration ids either directly from the $Recon$ service, or from other $RW$ processes. The value of $cmap$ is always in $Usable$, that is, $\pm$ for some finite prefix of $\mathbb{N}$, followed by an element of $C$, followed by elements of $C_\perp$, with only finitely many elements of $C$. When $RW_i$ processes a read or write operation, it uses all the configurations whose ids appear in its $cmap$, up to the first $\perp$.

The $pnum1$ variable and $pnum2$ array are used to implement a handshake that identifies "recent" messages. $RW_i$ uses $pnum1$ to count the total number of operation phases (either query or propagation phases) that it has initiated overall, including phases occurring in read, write, and garbage-collection operations. For every $j$, including $j = i$, $RW_i$ uses $pnum2(j)$ to record the largest number of a phase that $i$ has learned that $j$ has started, via a message from $j$ to $i$. Finally, two records, $op$ and $gc$, are used to maintain information about locally-initiated read, write, and garbage-collection operations.

**Joining and failure transitions.** When a $\mathsf{join}(\mathsf{rw})_i$ input occurs when $status = \mathsf{idle}$, if $i$ is the object's creator $i_0$, then $status$ immediately becomes active, which means that $RW_i$ is ready for full participation in the protocol. Otherwise, $status$ becomes joining, which means that $RW_i$ is receptive to inputs but not ready to perform any locally controlled actions. In either case, $RW_i$ records itself as a member of its own $world$. From this point on, $RW_i$ adds to its $world$ any process from which it receives a join message. (Recall that these join messages are sent by $Joiner$ automata.)

If $status = \mathsf{joining}$, then $status$ becomes active when $RW_i$ receives a message from another $RW$ process. (The code for this appears in the recv transition definition in Figure 5.) At this point, process $i$ has acquired enough information to begin participating fully. After $status$ becomes active, process $i$ can perform a $\mathsf{join\text{-}ack}(\mathsf{rw})$.

**Information propagation transitions.** Information is propagated between $RW$ processes in the background, via point-to-point channels that are accessed using the send and recv actions. The algorithm uses one kind of message, which contains a tuple consisting of the sender's $world$, its latest known $value$ and $tag$, its $cmap$, and two phase numbers—the current phase number of the sender, $pnum1$, and the latest known phase number of the receiver, from the $pnum2$ array. These messages may be sent at any time, to processes in the sender's $world$.

When $RW_i$ receives a message, it sets its $status$ to active, if it has not already done so. It adds incoming world information, in $W$, to its $world$ set. It compares the incoming tag $t$ to its own $tag$. If $t$ is strictly greater, it represents a more recent version of the object; in this case, $RW_i$ sets its $tag$ to $t$ and its $value$ to the incoming value $v$. $RW_i$ also updates its $cmap$ with the information in the incoming $CMap$, $cm$, using the $update$ operator

Input $\mathsf{join(rw)}_i$
Effect:
  if $\neg failed$ then
    if $status = \mathsf{idle}$ then
      if $i = i_0$ then
        $status \leftarrow \mathsf{active}$
      else
        $status \leftarrow \mathsf{joining}$
      $world \leftarrow world \cup \{i\}$

Input $\mathsf{recv(join)}_{j,i}$
Effect:
  if $\neg failed$ then
    if $status \neq \mathsf{idle}$ then
      $world \leftarrow world \cup \{j\}$

Output $\mathsf{join\text{-}ack(rw)}_i$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
Effect:
  none

Input $\mathsf{fail}_i$
Effect:
  $failed \leftarrow \mathsf{true}$

**Fig. 4.** $RW_i$: Join-related and failure transitions

defined in Section 2. That is, for each $k$, if $cmap(k) = \perp$ and $cm(k) \in C$, process $i$ sets its $cmap(k)$ to $cm(k)$. Also, if $cmap(k) \in C_\perp$ and $cm(k) = \pm$, indicating that the sender knows that configuration $k$ has already been garbage-collected, then $RW_i$ sets its $cmap(k)$ to $\pm$. $RW_i$ also updates its $pnum2(j)$ component for the sender $j$ to reflect new information about $j$'s phase number, which appears in the $pns$ component of the message.

While $RW_i$ is conducting a phase of a read, write, or garbage-collection operation, it verifies that the incoming message is "recent", in the sense that the sender $j$ sent it after $j$ received a message from $i$ that was sent after $i$ began the current phase. $RW_i$ uses the phase numbers to perform this check: it checks that the incoming phase number $pnr$ is at least as large as the current operation phase number ($op.pnum$ or $gc.pnum$). If the message is recent, then $RW_i$ uses it to update the $op$ or $gc$ record.

**Read and write operations.** A read or write operation is performed in two phases: a query phase and a propagation phase. In each phase, $RW_i$ obtains recent $value$, $tag$, and $cmap$ information from "enough" processes. This information is obtained by sending and receiving messages, as described above.

When $RW_i$ starts a phase of a read or write, it sets $op.cmap$ to a $CMap$ whose configurations are to be used to conduct the phase. Specifically, $RW_i$ uses $truncate(cmap)$, which is defined to include all the configuration ids in $cmap$ up to the first $\perp$. When a new $CMap$ $cm$ is received during the phase, $op.cmap$ is "extended" by adding all newly-discovered configuration ids, up to the first $\perp$ in $cm$. If adding these new configuration ids does not create a "gap", that is, if the extended $op.cmap$ is in $Truncated$, then the phase continues using the extended $op.cmap$. On the other hand, if adding these new configuration ids creates a gap, then $RW_i$ can infer that it has been using out-of-date configuration ids. In this case, it restarts the phase using the best currently known $CMap$, which is obtained by computing $truncate(cmap)$ for the latest local $cmap$.

In between restarts, while $RW_i$ is engaged in a single attempt to complete a phase, it never removes a configuration id from $op.cmap$. In particular, if process $i$ learns during a phase that a configuration id in $op.cmap(k)$ has been garbage-collected, it does not remove it from $op.cmap$, but continues to include it in conducting the phase.

Output $\mathsf{send}(\langle W, v, t, cm, pns, pnr\rangle)_{i,j}$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $j \in world$
  $\langle W, v, t, cm, pns, pnr\rangle =$
    $\langle world, value, tag, cmap, pnum1, pnum2(j)\rangle$
Effect:
  none

Input $\mathsf{recv}(\langle W, v, t, cm, pns, pnr\rangle)_{j,i}$
Effect:
  if $\neg failed$ then
    if $status \neq \mathsf{idle}$ then
      $status \leftarrow \mathsf{active}$
      $world \leftarrow world \cup W$
      if $t > tag$ then $(value, tag) \leftarrow (v, t)$
      $cmap \leftarrow update(cmap, cm)$
      $pnum2(j) \leftarrow \max(pnum2(j), pns)$
      if $op.phase \in \{\mathsf{query}, \mathsf{prop}\}$ and $pnr \geq op.pnum$ then
        $op.cmap \leftarrow extend(op.cmap, truncate(cm))$
        if $op.cmap \in Truncated$ then
          $op.acc \leftarrow op.acc \cup \{j\}$
        else
          $op.acc \leftarrow \emptyset$
          $op.cmap \leftarrow truncate(cmap)$
      if $gc.phase \in \{\mathsf{query}, \mathsf{prop}\}$ and $pnr \geq gc.pnum$ then
        $gc.acc \leftarrow gc.acc \cup \{j\}$

Input $\mathsf{new\text{-}config}(c, k)_i$
Effect:
  if $\neg failed$ then
    if $status \neq \mathsf{idle}$ then
      $cmap(k) \leftarrow update(cmap(k), c)$

Input $\mathsf{read}_i$
Effect:
  if $\neg failed$ then
    if $status \neq \mathsf{idle}$ then
      $pnum1 \leftarrow pnum1 + 1$
      $\langle op.pnum, op.type, op.phase, op.cmap, op.acc\rangle$
        $\leftarrow \langle pnum1, \mathsf{read}, \mathsf{query}, truncate(cmap), \emptyset\rangle$

Input $\mathsf{write}(v)_i$
Effect:
  if $\neg failed$ then
    if $status \neq \mathsf{idle}$ then
      $pnum1 \leftarrow pnum1 + 1$
      $\langle op.pnum, op.type, op.phase, op.cmap, op.acc,$
       $op.value\rangle$
        $\leftarrow \langle pnum1, \mathsf{write}, \mathsf{query}, truncate(cmap), \emptyset, v\rangle$

Internal $\mathsf{query\text{-}fix}_i$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $op.type \in \{\mathsf{read}, \mathsf{write}\}$
  $op.phase = \mathsf{query}$
  $\forall k \in \mathbb{N}, c \in C : (op.cmap(k) = c)$
    $\Rightarrow (\exists R \in read\text{-}quorums(c) :$
      $R \subseteq op.acc)$
Effect:
  if $op.type = \mathsf{read}$ then
    $op.value \leftarrow value$
  else
    $value \leftarrow op.value$
    $tag \leftarrow \langle tag.seq + 1, i\rangle$
  $pnum1 \leftarrow pnum1 + 1$
  $op.pnum \leftarrow pnum1$
  $op.phase \leftarrow \mathsf{prop}$
  $op.cmap \leftarrow truncate(cmap)$
  $op.acc \leftarrow \emptyset$

Internal $\mathsf{prop\text{-}fix}_i$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $op.type \in \{\mathsf{read}, \mathsf{write}\}$
  $op.phase = \mathsf{prop}$
  $\forall k \in \mathbb{N}, c \in C : (op.cmap(k) = c)$
    $\Rightarrow (\exists W \in write\text{-}quorums(c) :$
      $W \subseteq op.acc)$
Effect:
  $op.phase = \mathsf{done}$

Output $\mathsf{read\text{-}ack}(v)_i$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $op.type = \mathsf{read}$
  $op.phase = \mathsf{done}$
  $v = op.value$
Effect:
  $op.phase = \mathsf{idle}$

Output $\mathsf{write\text{-}ack}_i$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $op.type = \mathsf{write}$
  $op.phase = \mathsf{done}$
Effect:
  $op.phase = \mathsf{idle}$

**Fig. 5.** $RW_i$: Read/write transitions

The query phase of a read or write operation terminates when a *query fixed point* is reached. This happens when $RW_i$ determines that it has received recent responses from some read-quorum of each configuration in its current $op.cmap$. Then $t$, defined to be $RW_i$'s $tag$ at the query fixed point, is at least as great as the $tag$ value that each process in each of these read-quorums had at the start of the query phase.

If the operation is a read, then $RW_i$ determines at this point that its current $value$ is the value to be returned to its client. However, before returning, $RW_i$ performs the

propagation phase, whose purpose is to make sure that "enough" $RW$ processes have acquired tags that are at least $t$. Again, the information is propagated in the background, and $op.cmap$ is managed as described above. The propagation phase ends once a *propagation fixed point* is reached, when $RW_i$ has received recent responses from some write-quorum of each configuration in the current $op.cmap$. When this occurs, the $tag$ of each process in each of these write-quorums is at least $t$.

Processing for a write operation starting with $\mathsf{write}(v)_i$ is similar to that for a read. The query phase is conducted exactly as for a read, but processing after the query fixed point is different: Suppose $t$, process $i$'s $tag$ at the query fixed point, is of the form $(n, j)$. Then $RW_i$ defines the tag for its write operation to be the pair $(n+1, i)$ and sets its $tag$ to $(n+1, i)$ and its $value$ to the new value $v$. Then it performs its propagation phase. Now the purpose of the propagation phase is to ensure that "enough" processes acquire tags that are at least as great as $(n+1, i)$. The propagation phase is conducted exactly as for a read.

---

Internal $\mathsf{gc}(k)_i$
Precondition:
$\neg failed$
$status = \mathsf{active}$
$gc.phase = \mathsf{idle}$
$cmap(k) \in C$
$cmap(k+1) \in C$
$k = 0$ or $cmap(k-1) = \pm$
Effect:
$pnum1 \leftarrow pnum1 + 1$
$\langle gc.pnum, gc.phase, gc.acc, gc.index \rangle$
$\leftarrow \langle pnum1, \mathsf{query}, \emptyset, k \rangle$

Internal $\mathsf{gc\text{-}query\text{-}fix}(k)_i$
Precondition:
$\neg failed$
$status = \mathsf{active}$
$gc.phase = \mathsf{query}$
$gc.index = k$
$\exists R \in read\text{-}quorums(cmap(k)) :$
$\exists W \in write\text{-}quorums(cmap(k)) : R \cup W \subseteq gc.acc$
Effect:
$pnum1 \leftarrow pnum1 + 1$
$\langle gc.pnum, gc.phase, gc.acc \rangle \leftarrow \langle pnum1, \mathsf{prop}, \emptyset \rangle$

Internal $\mathsf{gc\text{-}prop\text{-}fix}(k)_i$
Precondition:
$\neg failed$
$status = \mathsf{active}$
$gc.phase = \mathsf{prop}$
$gc.index = k$
$\exists W \in write\text{-}quorums(cmap(k+1)) :$
$W \subseteq gc.acc$
Effect:
$cmap(k) \leftarrow \pm$

Internal $\mathsf{gc\text{-}ack}(k)_i$
Precondition:
$\neg failed$
$status = \mathsf{active}$
$gc.index = k$
$cmap(k) = \pm$
Effect:
$gc.phase = \mathsf{idle}$

---

**Fig. 6.** $RW_i$: Garbage-collection transitions

**New configurations and garbage collection.** When $RW_i$ learns about a new configuration id via a new-config input action, it simply records it in its $cmap$. From time to time, configuration ids get garbage-collected at $i$, in numerical order. The configuration ids used in performing query and propagation phases of reads and writes are those in $truncate(cmap)$, that is, all configurations that have not been garbage-collected and that appear before the first $\perp$.

There are two situations in which $RW_i$ may garbage-collect the configuration id in $cmap(k)$. First, $RW_i$ can garbage-collect $cmap(k)$ if it learns that another process has already garbage-collected it. This happens when $RW_i$ receives a message in which $cm(k) = \pm$. Second, $RW_i$ may acquire enough information to garbage-collect con-

figuration $k$ on its own. $RW_i$ accomplishes this by performing a a two-phase garbage-collection operation, with a structure similar to the read and write operations. $RW_i$ may initiate garbage-collection of configuration $k$ when its $cmap(k)$ and $cmap(k+1)$ are both in $C$, and when any configurations with indices smaller than $k-1$ have already been garbage-collected. Garbage-collection may proceed concurrently with a read or write operation at the same node.

In the query phase of a garbage-collection operation, $RW_i$ communicates with a read-quorum and a write-quorum of configuration $k$. The query phase accomplishes two tasks: First, $RW_i$ ensures that all the processes in the read-quorum and write-quorum learn about configurations $k$ and $k+1$, and also learn that all configurations smaller than $k$ have been garbage-collected. If such a process, $j$, is contacted afterwards by someone who is using configuration $k$, $j$ can tell that process about configuration $k+1$. Second, in the query phase, $RW_i$ collects $tag$ and $value$ information from the read-quorum and write-quorum. This ensures that, by the end of the query phase, $RW_i$'s $tag, t$, is at least as great as the $tag$ that each of the quorum members had when it sent a message to $RW_i$ for the query phase. In the propagation phase, $RW_i$ ensures that all the processes in a write-quorum of configuration $k+1$ have acquired $tag$s that are at least $t$. Note that, unlike a read or write operation, a garbage-collection for $k$ uses only two configurations—$k$ in the query phase and $k+1$ in the propagation phase.

At any time while $RW_i$ is garbage-collecting configuration $k$, it may discover that someone has already garbage-collected $k$; it discovers this by observing that $cmap(k) = \pm$. When this happens, $RW_i$ may simply terminate its garbage-collection.

### 5.3 The Complete Algorithm

We assume point to point channels $Channel_{i,j}$, one for each $i, j \in I$ (including $i = j$). $Channel_{i,j}$ is accessed using $\mathsf{send}(m)_{i,j}$ input actions, by which a sender at location $i$ submits message $m$ to the channel, and $\mathsf{recv}(m)_{i,j}$ output actions, by which a receiver at location $j$ receives $m$. Channels may lose and reorder messages, but cannot manufacture new messages or duplicate messages. Formally, we model $Channel_{i,j}$ as a multiset, where A $\mathsf{send}(m)_{i,j}$ input action adds one copy of $m$ to the multiset and A $\mathsf{recv}(m)_{i,j}$ output removes one copy of $m$. A lose input action allows any sub-multiset of messages to be removed.

The complete implementation, which we call $\mathcal{S}$, is the composition of the $Joiner_i$, $RW_i$, and $Channel_{i,j}$ automata, and any automaton whose traces satisfy the $Recon$ safety specification, with all actions that are not external actions of RAMBO hidden.

## 6 Safety Proof

We show that $\mathcal{S}$ satisfies the safety guarantees of RAMBO, as given in Section 3, assuming the environment safety assumptions. An *operation* can be of type read, write, or garbage-collection. An operations is uniquely identified by its starting event: $\mathsf{read}_i$, $\mathsf{write}(v)_i$, or $\mathsf{gc}(*)_i$ event.

We introduce the following history variables: (1) For every $k \in \mathbb{N}$: $c(k) \in C$. This is set when the first $\mathsf{new\text{-}config}(*, k)_*$ occurs, to the configuration id that appears as the

first parameter of this action. (2) For every operation $\pi$: $tag(\pi) \in T$. This is set just after $\pi$'s query-fix or gc-query-fix event, to the $tag$ of the process running $\pi$. (3) For every read or write operation $\pi$: (a) $query\text{-}cmap(\pi)$, a $CMap$. This is set in the query-fix step of $\pi$, to the value of $op.cmap$ in the pre-state. (b) $prop\text{-}cmap(\pi)$, a $CMap$. This is set in the prop-fix step of $\pi$, to the value of $op.cmap$ in the pre-state.

For any read or write operation $\pi$, we designate the following events: (1) query-phase-start($\pi$). This is defined in the query-fix step of $\pi$, to be the unique earlier event at which the collection of query results was started and not subsequently restarted (that is, $op.acc$ is set to $\emptyset$ in the effects of the corresponding step, and $op.acc$ is not later reset to $\emptyset$ following that event and prior to the query-fix step). This is either a read, write, or recv event. (2) prop-phase-start($\pi$). This is defined in the prop-fix step of $\pi$, to be the unique earlier event at which the collection of propagation results was started and not subsequently restarted. This is either a query-fix or recv event.

Now we present several lemmas describing information flow between operations. All are stated for a generic execution $\alpha$ satisfying the environment assumptions. The first lemma describes information flow between garbage-collection operations. We say that a gc-prop-fix$(k)_i$ event is *initial* if it is the first gc-prop-fix$(k)_*$ event in $\alpha$, and a garbage-collection operation is *initial* if its gc-prop-fix event is initial. The algorithm ensures that garbage-collection of successive configurations is sequential, in fact, for each $k$, the initial gc-prop-fix$(k)$ event precedes any attempt to garbage-collect $k+1$. Sequential garbage-collection implies that tags of garbage-collection operations are monotone with respect to the configuration indices:

**Lemma 1.** *Suppose $\gamma_k$ and $\gamma_\ell$ are garbage-collection operations for $k$ and $\ell$, respectively, where $k \leq \ell$ and $\gamma_k$ is initial. Suppose a* gc-query-fix$(\ell)$ *event for $\gamma_\ell$ occurs in $\alpha$. Then $tag(\gamma_k) \leq tag(\gamma_\ell)$.*

*Proof.* By induction on $\ell$, for fixed $k$. For the inductive step, assume that $\ell \geq k+1$ and the result is true for $\ell - 1$. A write-quorum of $c(\ell)$ is used in the propagation phase of $\gamma_{\ell-1}$ and a read-quorum of $c(\ell)$ is used in the query phase of $\gamma_\ell$. The quorum intersection property for $c(\ell)$ guarantees propagation of tag information. □

The following lemma describes situations in which certain configurations must appear in the $query\text{-}cmap$ of a read or write operation $\pi$. First, if no garbage-collection operation for $k$ completes before the query-phase-start event of $\pi$, then some configuration with index $\leq k$ must be included in $query\text{-}cmap(\pi)$. Second, if some garbage-collection for $k$ completes before the query-phase-start event of $\pi$, then some configuration with index $\geq k+1$ must be included in the $query\text{-}cmap(\pi)$.

**Lemma 2.** *Let $\pi$ be a read or write operation whose* query-fix *event occurs in $\alpha$. (1) If no* gc-prop-fix$(k)$ *event precedes the* query-phase-start$(\pi)$ *event, then $query\text{-}cmap(\pi)(\ell) \in C$ for some $\ell \leq k$. (2) If some* gc-prop-fix$(k)$ *event precedes the* query-phase-start$(\pi)$ *event, then $query\text{-}cmap(\pi)(\ell) \in C$ for some $\ell \geq k+1$.*

The next lemma describes propagation of $tag$ information from a garbage-collection operation to a following read or write operation.

**Lemma 3.** *Let $\gamma$ be an initial garbage-collection operation for $k$. Let $\pi$ be a read or write operation whose* query-fix *event occurs in $\alpha$. Suppose that the* gc-prop-fix$(k)$ *event of $\gamma$ precedes the* query-phase-start$(\pi)$ *event. Then $tag(\gamma) \leq tag(\pi)$, and if $\pi$ is a write operation then $tag(\gamma) < tag(\pi)$.*

The next two lemmas describe relationships between reads and writes that execute sequentially. The first lemma says that the smallest configuration index used in the propagation phase of the first operation is less than or equal to the largest index used in the query phase of the second operation. In other words, the second operation's query phase cannot use only configurations with indices that are less than any used in the first operation's propagation phase.

**Lemma 4.** *Assume $\pi_1$ and $\pi_2$ are two read or write operations such that the* prop-fix *event of $\pi_1$ precedes the* query-phase-start$(\pi_2)$ *event in $\alpha$.*
*Then* $\min(\{\ell : prop\text{-}cmap(\pi_1)(\ell) \in C\}) \leq \max(\{\ell : query\text{-}cmap(\pi_2)(\ell) \in C\})$.

*Proof.* Suppose not. Let $k = \max(\{\ell : query\text{-}cmap(\pi_2)(\ell) \in C\})$. Then some gc-prop-fix$(k)$ event occurs before the prop-fix of $\pi_1$, and so before the query-phase-start$(\pi_2)$ event. Lemma 2, Part 2, then implies that $query\text{-}cmap(\pi_2)(\ell) \in C$ for some $\ell \geq k+1$, which contradicts the choice of $k$. □

The second lemma describes propagation of $tag$ information between sequential reads and writes.

**Lemma 5.** *Suppose $\pi_1$ and $\pi_2$ are two read or write operations, such that the* prop-fix *event of $\pi_1$ precedes the* query-phase-start$(\pi_2)$ *event in $\alpha$. Then $tag(\pi_1) \leq tag(\pi_2)$, and if $\pi_2$ is a write then $tag(\pi_1) < tag(\pi_2)$.*

*Proof.* Let $i_1$ and $i_2$ be the processes that run operations $\pi_1$ and $\pi_2$, respectively. Let $cm_1 = prop\text{-}cmap(\pi_1)$ and $cm_2 = query\text{-}cmap(\pi_2)$. If there exists $k$ such that $cm_1(k) \in C$ and $cm_2(k) \in C$, then the quorum intersection property for configuration $k$ implies the conclusions of the lemma. So we assume that no such $k$ exists. Lemma 4 implies that $\min(\{\ell : cm_1(\ell) \in C\}) \leq \max(\{\ell : cm_2(\ell) \in C\})$. Since the set of indices used in each phase consists of consecutive integers and the intervals have no indices in common, it follows that $k_1 < k_2$, where $k_1 = \max(\{\ell : cm_1(\ell) \in C\})$ and $k_2 = \min(\{\ell : cm_2(\ell) \in C\})$.

Since, for every $k \leq k_2 - 1$, $query.cmap(\pi_2)(k) \notin C$, Lemma 2, Part 1, implies that, for every such $k$, a gc-prop-fix$(k)$ event occurs before the query-phase-start$(\pi_2)$ event. For each such $k$, define $\gamma_k$ to be the initial garbage-collection operation for $k$.

The propagation phase of $\pi_1$ accesses a write-quorum of $c(k_1)$, and the query phase of $\gamma_{k_1}$ accesses a read-quorum of $c(k_1)$. By the quorum intersection property, there is some $j$ in the intersection of these quorums. Let message $m$ be the message sent from $j$ to $i_1$ in the propagation phase of $\pi_1$, and let $m'$ be the message sent from $j$ to the process running $\gamma_{k_1}$ in its query phase. We claim that $j$ sends $m$ before it sends $m'$. For if not, then information about configuration $k_1 + 1$ would be conveyed by $j$ to $i_1$, who would include it in $cm_1$, contradicting the choice of $k_1$. Since $j$ sends $m$ before it sends $m'$, $j$ conveys $tag$ information from $\pi_1$ to $\gamma_{k_1}$, ensuring that $tag(\pi_1) \leq tag(\gamma_{k_1})$.

Since $k_1 \leq k_2 - 1$, Lemma 1 implies that $tag(\gamma_{k_1}) \leq tag(\gamma_{k_2-1})$. Lemma 3 implies that $tag(\gamma_{k_2-1}) \leq tag(\pi_2)$, and if $\pi_2$ is a write then $tag(\gamma_{k_2-1}) < tag(\pi_2)$. Combining all the inequalities then yields both conclusions. □

**Theorem 1.** *Let $\beta$ be a trace of $\mathcal{S}$. If $\beta$ satisfy the* RAMBO *environment assumptions, then $\beta$ satisfies the* RAMBO *service guarantees (well-formedness and atomicity).*

*Proof.* Let $\beta$ be a trace of $\mathcal{S}$ that satisfies the RAMBO environment assumptions. We argue that $\beta$ satisfies the RAMBO service guarantees. The proof that $\beta$ satisfies the RAMBO well-formedness guarantees is straightforward from the code. To show that $\beta$ satisfies atomicity (as defined in Section 3), assume that all read and write operations complete in $\beta$. Let $\alpha$ be an execution of $\mathcal{S}$ that satisfies the environment assumptions and whose trace is $\beta$. Define a partial order $\prec$ on read and write operations in $\alpha$: totally order the writes in order of their tags, and order each read with respect to all the writes so that a read with tag $t$ is ordered after all writes with tags $\leq t$ and before all writes with tags $> t$. Then we claim that $\prec$ satisfies the four conditions in the definition of atomicity. The interesting condition is Condition 2; the other three are straightforward.

For Condition 2, suppose for the sake of contradiction that $\pi_1$ and $\pi_2$ are read or write operations, $\pi$ completes before $\pi_2$ starts, and $\pi_2 \prec \pi_1$. If $\pi_2$ is a write operation, then since $\pi_1$ completes before $\pi_2$ starts, Lemma 5 implies that $tag(\pi_2) > tag(\pi_1)$. But the fact that $\pi_2 \prec \pi_1$ implies that $tag(\pi_2) \leq tag(\pi_1)$, yielding a contradiction. On the other hand, if $\pi_2$ is a read operation, then since $\pi_1$ completes before $\pi_2$ starts, Lemma 5 implies that $tag(\pi_2) \geq tag(\pi_1)$. But the fact that $\pi_2 \prec \pi_1$ implies that $tag(\pi_2) < tag(\pi_1)$, again yielding a contradiction. □

## 7 Implementation of the Reconfiguration Service

The *Recon* algorithm is considerably simpler than the $RW$ algorithm. It consists of a $Recon_i$ automaton for each location $i$, which interacts with a collection of global consensus services $Cons(k, c)$, one for each $k \geq 1$ and each $c \in C$, and with a point-to-point communication service.

$Cons(k, c)$ accepts inputs from members of configuration $c$, which it assumes to be the $k - 1^{st}$ configuration. These inputs are of the form $\mathsf{init}(c')_{k,c,i}$, where $c'$ is a proposed new configuration. The configuration that $Cons(k, c)$ decides upon (using $\mathsf{decide}(c')_{k,c,i}$ outputs) is deemed to be the $k^{th}$ configuration. The validity property of consensus implies that this decision is one of the proposed configurations.

$Recon_i$ is activated by a $\mathsf{join}(\mathsf{recon})_i$ action, which is an output of $Joiner_i$. $Recon_i$ accepts reconfiguration requests from clients, and initiates consensus to help determine new configurations. It records the new configurations that the consensus services determine. $Recon_i$ also informs $RW_i$ about newly-determined configurations, and disseminates information about newly-determined configurations to the members of those configurations. It returns acknowledgments and configuration reports to its client.

We implement $Cons(k, c)$ using the Paxos consensus algorithm [17], as described formally in [8]. Our complete implementation of $Recon$, $Recon_{impl}$, consists of the

$Recon_i$ automata, channels connecting all the $Recon_i$ automata, and the implementations of the $Cons$ services. We use the same kinds of channels as for RAMBO: point-to-point channels that may lose and reorder messages, but not manufacture new messages or duplicate messages. The complete RAMBO system (for a particular object) consists of $Joiner$, $RW$, and channel automata as described in Section 5, plus $Recon_{impl}$. We call the complete RAMBO system $\mathcal{S}'$.

## 8 Conditional Performance Analysis: Latency Bounds

We prove latency bounds for the full system $\mathcal{S}'$. To handle timing, we convert all the I/O automata to general timed automata (GTAs) as defined in [18], by allowing arbitrary amounts of time to pass in any state. Fix $d > 0$, the *normal message delay*, and $\epsilon > 0$.

RAMBO allows sending of messages at arbitrary times. For the purpose of latency analysis, we restrict RAMBO's sending pattern: We assume that each automaton has a local real-valued clock, and sends messages at the first possible time and at regular intervals of $d$ thereafter, as measured on the local clock. Also, non-send locally controlled events occur just once, within time 0 on the local clock.

Our results also require restrictions on timing and failure behavior: We define an admissible timed execution to be *normal* provided that all local clocks progress at rate exactly 1, all messages that are sent are delivered within time $d$, and timing and failure behavior for all consensus services is "normal", as defined in [8].[2]

Next, we define a reliability property for configurations. In general, in quorum-based systems, operations that use quorums are guaranteed to terminate only if some quorums do not fail. Because we use many configurations, we attempt to take into account which configurations might be in use. We say that $k$ is *installed* in a timed execution $\alpha$ provided that either $k = 0$ or there exists $c \in C$ such that (1) some $\mathsf{init}(*)_{k,c,*}$ event occurs, and (2) for every $i \in members(c)$, either $\mathsf{decide}(*)_{k,c,i}$ or $\mathsf{fail}_i$ occurs. (Thus, configuration $k-1$ is $c$, and every non-failed member of $c$ has learned about configuration $k$.) We say that $\alpha$ is *$e$-configuration-viable*, $e \geq 0$, provided that for every $c$ and $k$ such that some $rec\text{-}cmap(k)_* = c$ in some state in $\alpha$, there exist $R \in read\text{-}quorums(c)$ and $W \in write\text{-}quorums(c)$ such that either (1) no process in $R \cup W$ ever fails in $\alpha$, or (2) $k+1$ is installed in a finite prefix $\alpha'$ of $\alpha$ and no process in $R \cup W$ fails in $\alpha$ by time $\ell time(\alpha') + e$. (Quorums remain non-failed for at least time $e$ after the next configuration is installed.)

The $e$-configuration-viability assumption is useful in situations where a configuration is no longer needed for performing operations after time $e$ after the next configuration is installed. This condition holds in RAMBO executions in which certain timing assumptions hold; the strength of those assumptions determines the value of $e$ that must be considered. We believe that such an assumption is reasonable for a reconfigurable system, because it can be reconfigured when quorums appear to be in danger of failing.

We prove a bound of $2d$ on the time to join, and a bound of $11d + \epsilon$ for the time for reconfiguration, based on a bound of $10d + \epsilon$ for consensus. We also establish a

[2] This means that all messages are delivered within time $d$, local processing time is 0, and information is gossiped at intervals of $d$.

situation in which a system is guaranteed to produce a positive response to a reconfiguration request. We prove a bound of $4d$ on the time for garbage-collection, assuming that enough of the relevant processes remain non-failed. We prove a bound of $4d$ on the latency for read and write operations in a "quiescent" situation, in which all joins and configuration management events have stopped, and the configuration map of the operation's initiator includes the latest configuration and has value $\pm$ for all earlier configurations. More generally, we show that this bound holds even if this map contains more than one configuration: since the configurations are used concurrently, the use of multiple configurations does not slow the operation down.

We show that all participants succeed in exchanging information about configurations, within a short time: if $i$ and $j$ have joined at least time $e$ ago and do not fail, then any information that $i$ has about configurations is conveyed to $j$ within time $2d$. Using this result, we show that, if reconfiguration requests are spaced sufficiently far apart, and if quorums of configurations remain alive for sufficiently long, then garbage collection keeps up with reconfiguration.

The main latency theorem bounds the time for read and write operations in the "steady-state" case, where reconfigurations do not stop, but are spaced sufficiently far apart. Fix $e \geq 0$.

**Theorem 2.** *Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$ such that:*
*(1) If a* $\mathsf{recon}(*, c)_i$ *event occurs at time $t$ then for every $j \in members(c)$,* $\mathsf{join\text{-}ack}(\mathsf{rambo})_j$ *occurs by time $t - e$. (2) If* $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ *and* $\mathsf{join\text{-}ack}(\mathsf{rambo})_j$ *both occur by time $t$, and neither $i$ nor $j$ fails by time $t+e$, then by time $t+e$, $i \in world_j$. (3) For any* $\mathsf{recon}(c, *)_i$ *that occurs in $\alpha$, the time since the corresponding* $\mathsf{report}(c)_i$ *event is $\geq 12d + \varepsilon$. (4) $\alpha$ satisfies $11d$-configuration-viability. (5) $\alpha$ contains* decide *events for infinitely many configurations.*
*Suppose that a* $\mathsf{read}_i$ *(resp.,* $\mathsf{write}(*)_i$*) event occurs at time $t$, and* $\mathsf{join\text{-}ack}_i$ *occurs strictly before time $t - (e + 8d)$. Then the corresponding* $\mathsf{read\text{-}ack}_i$ *(resp.,* $\mathsf{write\text{-}ack}(*)_i$*) event occurs by time $t + 8d$.*

*Proof.* The various spacing properties and bounds on time to disseminate information imply that each phase of the read or write completes with at most one restart for learning about a new configuration. Therefore, each phase takes time at most $4d$, for a total of $8d$. □

In the full paper we also present latency results analogous to those described above, for executions that have normal timing and failure characteristics after some point in the execution. These results are similar to the previous results, but include dependence on the time when normal behavior begins.

## 9 Future Work

In future work, we plan to implement the complete $Rambo$ algorithm in LAN, WAN, and mobile settings. We will extend our performance analysis and compare it with empirical results. We will investigate ways of increasing the concurrency of garbage-collection and of reducing the amount of communication. Finally, this work leaves open the question of how to choose good configurations, for various kinds of platforms.

## References


1. *Communications of the ACM*, special section on group communications, vol. 39, no. 4, 1996.
2. Y. Amir, D. Dolev, P. Melliar-Smith and L. Moser, “Robust and Efficient Replication Using Group Communication” Tech. Rep. 94-20, Dept. of Computer Science, Hebrew Univ., 1994.
3. H. Attiya, A. Bar-Noy and D. Dolev, “Sharing Memory Robustly in Message Passing Systems", *J. of the ACM*, vol. 42, no. 1, pp. 124-142, 1996.
4. P.A. Bernstein, V. Hadzilacos and N. Goodman, “Concurrency Control and Recovery in Database Systems”, Addison-Wesley, Reading, MA, 1987.
5. F. Cristian and F. Schmuck, “Agreeing on Processor Group Membership in Asynchronous Distributed Systems”, TR. CSE95-428, Dept. of Comp. Sci., Univ. of California San Diego.
6. S.B. Davidson, H. Garcia-Molina and D. Skeen, “Consistency in Partitioned Networks", *ACM Computing Surveys*, vol. 15, no. 3, pp. 341-370, 1985.
7. R. De Prisco, A. Fekete, N. Lynch, A. Shvartsman, “A Dynamic Primary Configuration Group Communication Service”, *13th Int-l Conference of Distributed Computing*, 1999.
8. Roberto De Prisco, Nancy Lynch, Alex Shvartsman, Nicole Immorlica and Toh Ne Win “A Formal Treatment of Lamport’s Paxos Algorithm”, manuscript, 2002.
9. C. Dwork, N. A. Lynch, L. J. Stockmeyer, “Consensus in the presence of partial synchrony”, *J. of ACM*, 35(2), pp. 288-323, 1988.
10. B. Englert and A.A. Shvartsman, Graceful Quorum Reconfiguration in a Robust Emulation of Shared Memory, in *Proc. International Conference on Distributed Computer Systems* (ICDCS’2000), pp. 454-463, 2000.
11. A. Fekete, N. Lynch and A. Shvartsman “Specifying and using a partitionable group communication service”, *ACM Trans. on Computer Systems*, vol. 19, no. 2, pp. 171–216, 2001.
12. H. Garcia-Molina and D. Barbara, “How to Assign Votes in a Distributed System," *J. of the ACM*, vol. 32, no. 4, pp. 841-860, 1985.
13. D.K. Gifford, “Weighted Voting for Replicated Data", in *Proc. of 7th ACM Symp. on Oper. Sys. Princ.*, pp. 150-162, 1979.
14. M.P. Herlihy, “Dynamic Quorum Adjustment for Partitioned Data", *ACM Trans. on Database Systems*, 12(2), pp. 170-194, 1987.
15. R. Guerraoui and A. Schiper, “Consensus Service: A Modular Approach For Building Fault-Tolerant Agreement Protocols in Distributed Systems", *Proc. of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pp. 168-177, 1996.
16. I. Keidar and D. Dolev, “Efficient Message Ordering in Dynamic Networks”, in *Proc. of 15th Annual ACM Symp. on Princ. of Distr. Comput.*, pp. 68-76, 1996.
17. Leslie Lamport, "The Part-Time Parliament", *ACM Transactions on Computer Systems*, 16(2) 133-169, 1998.
18. N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.
19. N. Lynch and A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. *27th Int-l Symp. on Fault-Tolerant Comp.*, pp. 272-281, 1997.
20. Nancy Lynch and Alex Shvartsman. RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. MIT-LCS-TR-856, 2002
21. E. Upfal and A. Wigderson, How to share memory in a distributed system, *Journal of the ACM*, 34(1):116–127, 1987.