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

Nancy Lynch† Alex Shvartsman‡

August 16, 2002

**Abstract**

This paper presents an algorithm that emulates atomic read/write shared objects in a dynamic network setting. To ensure that the data is highly available and long-lived, each object is replicated at several network locations. 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 are allowed to change during computation, and such changes do not cause violations of atomicity. Any quorum configuration may be installed at any time—no intersection requirement is imposed on the sets of members or on the quorums of distinct configurations. The algorithm tolerates processor stopping failures and message loss.

The algorithm performs three major activities, all concurrently: (1) reading and writing objects, (2) choosing new configurations and notifying members, and (3) identifying and removing ("garbage-collecting") obsolete configurations. The algorithm is composed of two sub-algorithms: a *main algorithm*, which handles reading, writing, and garbage-collection, and a *reconfiguration algorithm*, which handles the selection and dissemination of new configurations.

The algorithm guarantees atomicity in the presence of arbitrary patterns of asynchrony and failures. The algorithm satisfies a variety of conditional performance properties, based on a variety of timing and failure assumptions. In particular, if participants gossip periodically in the background, if garbage-collection is scheduled periodically, if reconfiguration is not requested too frequently, and if quorums of active configurations do not fail, then read and write operations complete within time $8d$, where $d$ is the maximum message latency.

*This work was supported in part by the NSF ITR Grant CCR-0121277.

†Massachusetts Institute of Technology, Laboratory for Computer Science, 200 Technology Square, NE43-365, Cambridge, MA 02139, USA. Email: `lynch@theory.lcs.mit.edu`. The work of this author was also supported by AFOSR under contract F49620-00-1-0097 and by NTT under contract MIT9904-12.

‡Department of Computer Science and Engineering, 191 Auditorium Road, Unit 3155, University of Connecticut, Storrs, CT 06269 and Massachusetts Institute of Technology, Laboratory for Computer Science, 200 Technology Square, NE43-316, Cambridge, MA 02139, USA. Email: `alex@theory.lcs.mit.edu`. The work of this author was also supported by NSF CAREER Award 9984778 and NSF Grant 9988304.

# 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.[1] 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 at several network locations. 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—no intersection requirement is imposed on the sets of members or on the quorums of distinct configurations.

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". (Here "Basic" means "Read/Write".) The rest of the paper presents our algorithm and its analysis. The algorithm carries out three major activities, all concurrently: (1) reading and writing objects, (2) choosing new configurations and notifying members, and (3) identifying 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 only loosely coupled to the main read-write algorithm, in particular, several configurations may be known to the algorithm at one time, and read and write operations can use them all without any harm.

The main algorithm performs read and write operations requested by clients using a two-phase strategy, where 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

[1] We do not explicitly consider participants leaving, but treat that case in the same way as a failure.

Paxos algorithm [30], as described formally in [14]. Although such consensus executions may be slow—in fact, in some situations, they may not even terminate—they do not cause any delays for read and write operations.

We specify all services and algorithms, and their interactions, using I/O automata. We show correctness (atomicity) of the algorithm for arbitrary patterns of asynchrony and failures. On the other hand, we analyze performance *conditionally*, based on certain failure and timing 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.

One possible application for the RAMBO service is for maintaining reliable information in a military mission. Data objects might represent the latest status information for various real-world entities, such as friendly and unfriendly vehicles and soldiers. Although all participants might need to read or write the values of such a data object, a small number of participants, for example, those currently operating in the geographical vicinity of the real-world entity, might assume responsibility for maintaining the object's integrity. In this case, it would be reasonable to change the configuration from time to time, based on which participants are currently in the vicinity.

**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, as suggested by Lamport [30]. In contrast, we use consensus to agree only on the sequence of configurations and not on the individual read and write operations. Since reaching consensus is costly, our approach leads to better performance for reads and writes. Also, in our algorithm, the termination of consensus affects the termination of reconfiguration attempts, but not of read and write operations: reads and writes are guaranteed to complete, provided that currently active configurations are not disabled by failures.

Group communication services (GCSs) [1] can also be used to implement an atomic data service in a dynamic network. This can be done, for example, by implementing a global totally ordered broadcast service on top of a view-synchronous GCS [19] using techniques of Amir, Dolev, Keidar, Melliar-Smith and Moser [28, 29, 5]. Our approach compares favorably with these implementations: In most GCS-based implementations, forming a new view following a crash takes a substantial amount of time, and client-level operations are delayed during the view-formation period. In contrast, although reconfiguration can be slow in our algorithm, reads and writes continue to make progress during reconfiguration. Also, in some standard GCS implementations, performance is degraded even if only one failure occurs. For example, in ring-based implementations like that of Cristian and Schmuck [10] a single failure triggers the formation of a new view. In contrast, our algorithm uses quorums to tolerate small numbers of failures.

De Prisco, Fekete, Lynch, and Shvartsman [13] introduced the notion of primary configurations and defined a dynamic primary configuration group communication service. They also showed how to implement dynamic atomic memory over such a service, using a version of the algorithm of Attiya, Bar-Noy, and Dolev [7] within each configuration. That work restricts the set of possible new configurations to those satisfying certain intersection properties with previous configurations, whereas we impose no such restrictions—we allow *arbitrary* new configurations to be installed. Like other solutions based on group communication, the algorithm of [13] delays reads and writes during reconfiguration.

In earlier work on atomic memory for dynamic networks, [34, 18], we considered *single reconfigurer* approaches, in which a single designated participant initiates all reconfiguration requests. This approach has the disadvantage that the failure of the single reconfigurer disables future reconfiguration. In contrast, in our new approach, any member of the latest configuration may propose the next configuration, and fault-tolerant consensus is used to ensure that a unique next configuration is determined. For well-chosen quorums, this approach avoids single points of failure: new configurations can continue to be produced, in spite of the failures of some of the configuration members. Another difference is that, in [34, 18], garbage-collection of an old configuration is tightly coupled to the introduction of a new configuration. Our new approach allows garbage-collection of old configurations to be carried out in the background, concurrently with other processing. A final difference is that, in [34, 18], information about new configurations is propagated only during the processing of read and write operations. A client who does not perform any operations for a long while may become "disconnected" from the latest configuration, if older configurations become disabled. In contrast, in our new algorithm, information about configurations is gossiped periodically, in the background, which permits all participants to learn about new configurations and garbage-collect old configurations.

**Other related work.** Upfal and Wigderson produced the first general scheme for emulating shared memory in message-passing systems by using replication and accessing majorities of timestamped replicas [39]. Attiya, Bar-Noy, and Dolev developed a majority-based emulation of atomic read/write memory [7]. Their 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.

Quorums [21] are generalizations of majorities. A *quorum system* is a collection of quorum sets such that any two quorums intersect [20]. Another approach is to classify quorums as read-quorums and write-quorums such that any read-quorum intersects any write-quorum, and (sometimes) such that any two write-quorums intersect. Quorums have been used to implement data replication protocols [2, 8, 9, 11, 16, 17, 22, 23].

Consensus algorithms have been used as building blocks in other work, e.g, [27].

**Paper organization.** The rest of the paper is organized as follows. Section 2 describes some data types used by our algorithms. Section 3 contains our specification for the Rambo reconfigurable atomic memory service. Section 4 contains the specification for the *Recon* reconfiguration service. Section 5 contains the main algorithm, assuming the *Recon* service, and Section 6 contains the proof that the algorithm satisfies the Rambo specification. Section 7 contains the algorithm to implement the *Recon* specification, using consensus. Section 8 contains the analysis of latency under "normal" timing and failure assumptions, and Section 9 contains the analysis of latency when normal behavior begins at some point in the execution. Finally, Section 10 contains our conclusions.

## 2 Data Types

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

We assume the following specific data types, distinguished elements, and functions.

- $I$, 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 $x \in X$:
  - $V_x$, the set of values that object $x$ may take on.
  - $(v_0)_x \in V_x$, the initial value of $x$.
- $C$, the set of *configuration identifiers*. We assume only the trivial partial order on $C$, in which all elements are incomparable; in the resulting augmented partial ordering of $C_\pm$, all elements of $C$ are still incomparable.
- For each $x \in X$, $(c_0)_x \in C$, the *initial configuration identifier* for $x$.
- For each $c \in C$ we define:
  - $\mathit{members}(c)$, a finite subset of $I$.
  - $\mathit{read\text{-}quorums}(c)$, a set of finite subsets of $\mathit{members}(c)$.
  - $\mathit{write\text{-}quorums}(c)$, a set of finite subsets of $\mathit{members}(c)$.

  We assume the following constraints:
  - $\mathit{members}((c_0)_x) = \{(i_0)_x\}$. That is, the initial configuration for object $x$ has only a single member, who is the creator of $x$.
  - For every $c$, every $R \in \mathit{read\text{-}quorums}(c)$, and every $W \in \mathit{write\text{-}quorums}(c)$, $R \cap W \neq \emptyset$.
- *update*, a binary function on $C_\pm$, defined by $\mathit{update}(c, c') = \max(c, c')$ if $c$ and $c'$ are comparable (in the augmented partial ordering of $C_\pm$), $\mathit{update}(c, c') = c$ otherwise.
- *extend*, a binary function on $C_\pm$, defined by $\mathit{extend}(c, c') = c'$ if $c = \perp$ and $c' \in C$, and $\mathit{extend}(c, c') = c$ otherwise.
- *CMap*, the set of *configuration maps*, defined as the set of mappings from $\mathbb{N}$ to $C_\pm$, $\mathbb{N} \to C_\pm$. We extend the *update* and *extend* operators elementwise to binary operations on *CMap*.
- *truncate*, a unary function on *CMap*, defined by $\mathit{truncate}(cm)(k) = \perp$ if there exists $\ell \leq k$ such that $cm(\ell) = \perp$, $\mathit{truncate}(cm)(k) = cm(k)$ otherwise. This truncates configuration map $cm$ by removing all the configuration identifiers that follow a $\perp$.
- *Truncated*, the subset of *CMap* such that $cm \in \mathit{Truncated}$ if and only if $\mathit{truncate}(cm) = cm$.

- *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 $\perp$.

**Lemma 2.1** *If $cm \in Usable$ then:*

1. *If $k, \ell \in \mathbb{N}$, $k \leq \ell$, and $cm(\ell) = \pm$, then $cm(k) = \pm$.*
2. *$cm$ contains finitely many $\pm$ entries.*
3. *$cm$ contains finitely many $C$ entries.*
4. *If $k \in \mathbb{N}$, $cm(k) = \pm$, and $cm(k+1) \neq \pm$, then $cm(k+1) \in C$.*

The following lemma says that various operations preserve the "usable" property:

**Lemma 2.2**

1. *If $cm, cm' \in Usable$ then $update(cm, cm') \in Usable$.*
2. *If $cm \in Usable$, $k \in N$, $c \in C$, and $cm'$ is identical to $cm$ except that $cm'(k) = update(cm(k), c)$, then $cm' \in Usable$.*
3. *If $cm, cm' \in Usable$ then $extend(cm, cm') \in Usable$.*
4. *If $cm \in Usable$ then $truncate(cm) \in Usable$.*

**Proof.** Part 1 is shown using a case analysis based on which of $cm$ and $cm'$ has a longer prefix of $\pm$s. Part 2 uses a case analysis based on where $k$ is with respect to the prefix of $\pm$s. Part 3 and Part 4 are also straightforward. $\square$

# 3 Reconfigurable Atomic Memory Service Specification

This section contains our specification for the RAMBO reconfigurable atomic memory service. This specification consists of an external signature (interface) plus a set of traces that embody RAMBO's safety properties. No liveness properties are included in the specification; we replace these with conditional latency bounds, which are stated and proved in Sections 8 and 9. The external signature appears in Figure 1. (We use I/O automata notation for all of our specifications.)

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 best 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 action. The client initiates a reconfiguration using a $\mathsf{recon}_i$ input action, which is acknowledged with a $\mathsf{recon\text{-}ack}_i$ output action. RAMBO reports a new configuration to the client using a $\mathsf{report}_i$ output action. Finally, a crash at location $i$ is modelled using a $\mathsf{fail}_i$ input action. We do not explicitly model graceful process "leaves", but instead we model process departures as failures.
Now we define the set of traces describing RAMBO's safety properties. These traces are defined to be those that satisfy an implication of the form "environment assumptions imply service guarantees". The environment assumptions are simple "well-formedness" conditions:

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

Figure 1: RAMBO($x$): External signature

- *Well-formedness:*
  - For every $x$ and $i$:
    - 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.
    - At most one $\mathsf{join}(\mathsf{rambo}, *)_{x,i}$ event occurs.
    - 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.
    - 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.
  - For every $x$ and $c$, at most one $\mathsf{recon}(*, c)_{x,*}$ event occurs.
    This says that configuration identifiers that are proposed in recon events are unique. It does not say that the membership and/or quorum sets are unique—just the identifiers. The same membership and quorum sets may be associated with different configuration identifiers.
  - 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
    - A $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,j}$ event for every $j \in members(c')$.

    This says participant $i$ can request reconfiguration from $c$ to $c'$ only if $i$ has previously receives a report that $c$ is the current configuration identifier, and only if all the members of $c'$ have already joined.

The safety guarantees provided by the service are as follows:

- *Well-formedness:* For every $x$ and $i$:
  - 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.
  - 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:*[2] 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:

[2] 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 [33]. Note that Lemma 13.16 of [33] is presented for a setting

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, in which uniquely identified asynchronous processes communicate using point-to-point channels. 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 $x$. Therefore, throughout the rest of the paper, we describe an implementation for a particular $x$, and (usually) suppress explicit mention of $x$. Thus, we write $V$, $v_0$, $c_0$, and $i_0$ from now on as shorthand for $V_x$, $(v_0)_x$, $(c_0)_x$, and $(i_0)_x$, respectively.

## 4 Reconfiguration Service Specification

Our RAMBO implementation for each object $x$ consists of a main *Reader-Writer* algorithm and a reconfiguration service, *Recon*$(x)$; since we are suppressing mention of $x$, we write this simply as *Recon*. In this section, we present the specification for the *Recon* service, as an external signature and a set of traces. We present our implementation of *Recon* in Section 7, after we present the main *Reader-Writer* algorithm and the proof of its safety properties.

The external signature for *Recon* 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 to reconfigure the object using a $\mathsf{recon}_i$ input, which is acknowledged with a $\mathsf{recon\text{-}ack}_i$ output action. RAMBO reports a new configuration to the client using a $\mathsf{report}_i$ output action. Crashes are modeled using $\mathsf{fail}$ actions.

*Recon* also produces outputs of the form $\mathsf{new\text{-}config}(c, k)_i$, which announce at location $i$ that $c$ is the $k^{th}$ configuration identifier for the object. These outputs are used for communication with the portion of the *Reader-Writer* algorithm running at location $i$. *Recon* announces consistent information, only one configuration identifier per index in the configuration identifier sequence. It delivers information about each configuration to members of the new configuration and of the immediately preceding configuration.

Now we define the set of traces describing *Recon*'s safety properties. Again, these are defined in terms of environment assumptions and and service guarantees. The environment assumptions are simple well-formedness conditions, consistent with the well-formedness assumptions for RAMBO:

with only finitely many locations, whereas we consider infinitely many locations. However, nothing in Lemma 13.16 or its proof depends on the finiteness of the set of locations, so the result carries over immediately to our setting. The other relevant results accompanying Lemma 13.16 also carry over to this setting; in particular, Theorem 13.1, which asserts that atomicity is a safety property, and Lemma 13.10, which asserts that it suffices to consider executions in which all operations complete, both carry over.

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

Figure 2: *Recon*: External signature

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

The safety guarantees provided by the service are as follows:

- *Well-formedness:* For every $i$:
  - 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.
  - 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 $c = c'$. (No disagreement arises about what the $k^{th}$ configuration identifier is, for any $k$.)
- *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. (Any configuration identifier that is announced was previously requested by someone who did not receive a negative acknowledgment.)
- *No duplication:* If $\mathsf{new\text{-}config}(c, k)_i$ and $\mathsf{new\text{-}config}(c, k')_j$ both occur, then $k = k'$. (The same configuration identifier cannot be assigned to two different positions in the identifier sequence.)

# 5 Implementation of RAMBO Using a Reconfiguration Service

Our implementation of RAMBO includes $\mathit{Joiner}_{x,i}$ automata for each $x$ and $i$, which handle joining of new participants, and $\mathit{Reader\text{-}Writer}_{x,i}$ automata, which handle reading, writing, and "installing" new configurations. The *Reader-Writer* and *Joiner* automata have access to asynchronous communication channels $\mathit{Channel}_{x,i,j}$. The *Reader-Writer* automata also interact with an arbitrary implementation of the *Recon* service. The architecture is depicted in Figure 3.

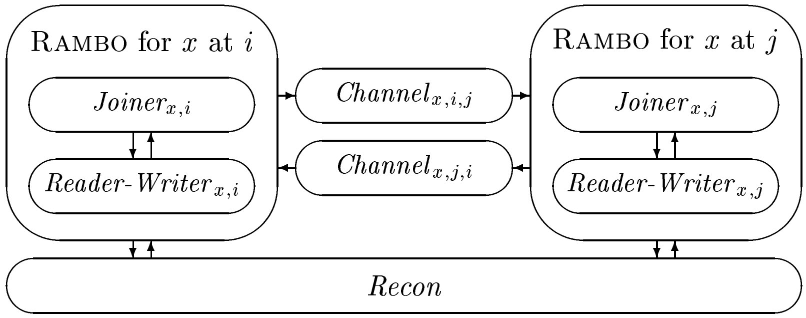


Figure 3: RAMBO architecture: The diagram depicts the *Joiner* and *Reader-Writer* automata at $i$ and $j$, the *Channel* automata, and the *Recon* service.

In this section we present the $Joiner_{x,i}$, $Reader\text{-}Writer_{x,i}$, and $Channel_{x,i,j}$ automata. As before, since we are suppressing explicit mention of $x$, we write simply $Joiner_i$, $Reader\text{-}Writer_i$, and $Channel_{i,j}$, leaving the object $x$ implicit.

## 5.1 Joiner automata

The joining protocol is implemented by a separate $Joiner_i$ automaton for each $i$. The signature, state and transitions of $Joiner_i$ all appear in Figure 4.

When $Joiner_i$ receives a $\mathsf{join}(\mathsf{rambo}, J)$ request from its environment, it carries out a simple protocol: It sends join messages to the processes in $J$ (with the hope that they are already participating, and so can help in the attempt to join). Also, it submits join requests to the local *Reader-Writer* and *Recon* components and waits for acknowledgments for these requests. The join messages that are sent by *Joiner* automata are not handled by *Joiner* automata at other locations, but rather, by *Reader-Writer* automata, as discussed in the next subsection.

## 5.2 Reader-Writer automata

The heart, and hardest part, of our RAMBO implementation is the reader-writer algorithm, which handles the processing of read and write operations. Each read or write operation is processed using one or more configurations, which it learns about from the *Recon* service. The reader-writer protocol also handles the garbage-collection of older configurations, which ensures that later read and write operations need not use them.

The reader-writer protocol is implemented by $Reader\text{-}Writer_i$ automaton for all $i$. The $Reader\text{-}Writer_i$ components interact with the *Recon* service and communicate using point-to-point asynchronous channels.

### 5.2.1 Signature and state

The signature and state of $Reader\text{-}Writer_i$ appear in Figure 5.

The state variables are used as follows. The *status* variable keeps track of the progress of the component as it joins the protocol. When $status = \mathsf{idle}$, $Reader\text{-}Writer_i$ does not respond to any inputs (except for join) and does not perform any locally controlled actions. When $status = \mathsf{joining}$,

**Signature:**

Input:
- $\mathsf{join}(\mathsf{rambo}, J)_i$, $J$ a finite subset of $I - \{i\}$
- $\mathsf{join\text{-}ack}(r)_i$, $r \in \{\mathsf{recon}, \mathsf{rw}\}$
- $\mathsf{fail}_i$

Output:
- $\mathsf{send}(\mathsf{join})_{i,j}$, $j \in I - \{i\}$
- $\mathsf{join}(r)_i$, $r \in \{\mathsf{recon}, \mathsf{rw}\}$
- $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$

**State:**

$status \in \{\mathsf{idle}, \mathsf{joining}, \mathsf{active}\}$, initially $\mathsf{idle}$
$child\text{-}status \in \{\mathsf{recon}, \mathsf{rw}\} \to \{\mathsf{idle}, \mathsf{joining}, \mathsf{active}\}$, initially everywhere $\mathsf{idle}$
$hints \subseteq I$, initially $\emptyset$
$failed$, a Boolean, initially $false$

**Transitions:**

Input $\mathsf{join}(\mathsf{rambo}, J)_i$
Effect:
  if $\neg failed$ then
    if $status = \mathsf{idle}$ then
      $status \leftarrow \mathsf{joining}$
      $hints \leftarrow J$

Output $\mathsf{send}(\mathsf{join})_{i,j}$
Precondition:
  $\neg failed$
  $status = \mathsf{joining}$
  $j \in hints$
Effect:
  none

Output $\mathsf{join}(r)_i$
Precondition:
  $\neg failed$
  $status = \mathsf{joining}$
  $child\text{-}status(r) = \mathsf{idle}$
Effect:
  $child\text{-}status(r) \leftarrow \mathsf{joining}$

Input $\mathsf{join\text{-}ack}(r)_i$
Effect:
  if $\neg failed$ then
    if $status = \mathsf{joining}$ then
      $child\text{-}status(r) \leftarrow \mathsf{active}$

Output $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$
Precondition:
  $\neg failed$
  $status = \mathsf{joining}$
  $\forall r \in \{\mathsf{recon}, \mathsf{rw}\} : child\text{-}status(r) = \mathsf{active}$
Effect:
  $status \leftarrow \mathsf{active}$

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

Figure 4: $Joiner_i$

$Reader\text{-}Writer_i$ is receptive 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 attempted 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 $cmap(k) = \bot$, it means that $Reader\text{-}Writer_i$ has not yet learned what the $k^{th}$ configuration identifier is. If $cmap(k) = c \in C$, it means that $Reader\text{-}Writer_i$ has learned that the $k^{th}$ configuration identifier is $c$, and it has not yet garbage-collected it. If $cmap(k) = \pm$, it means that $Reader\text{-}Writer_i$ has garbage-collected the $k^{th}$ configuration identifier. $Reader\text{-}Writer_i$ learns about configuration identifiers either directly, from the *Recon* service, or indirectly, from other *Reader-Writer* processes. The value of *cmap* is always in *Usable*, that is, $\pm$ for some finite (possibly zero length) prefix of $\mathbb{N}$, followed by an element of $C$, followed by elements of $C_\bot$, with only finitely many total elements of

**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 $\mathsf{idle}$
$world$, 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 $\mathsf{idle}$
  $pnum \in \mathbb{N}$
  $cmap \in CMap$
  $acc$, a finite subset of $I$
  $value \in V$

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

Figure 5: *Reader-Writer$_i$*: Signature and state

$C$. When *Reader/Writer$_i$* processes a read or write operation, it uses all the configurations whose identifier 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. *Reader-Writer$_i$* uses *pnum1* to count the total number of operation "phases" it has initiated overall, including phases occurring in read, write, and garbage-collection operations. (A "phase" here refers to either a query or propagate phase, as described below.) For every $j$, including $j = i$, *Reader-Writer$_i$* uses $pnum2(j)$ to record the largest number of a phase that $i$ has learned that $j$ has started, via a direct message from $j$ to $i$. Finally, two records, *op* and *gc*, are used to maintain information about a locally-initiated read, write, or garbage-collection operation in progress.

### 5.2.2 Transitions

The transitions are presented in three figures: Figure 6 presents the transitions pertaining to joining the protocol and failing. Figure 7 presents those pertaining to reading and writing, and Figure 8 presents those pertaining to garbage-collection.

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

If *status* = joining, then *status* becomes active when *Reader-Writer*$_i$ receives a message from another process. (The code for this appears in the recv transition definition in Figure 7.) At this point, process $i$ has acquired enough information to begin participating fully. After *status* becomes active, process $i$ can perform a join-ack(rw).

---

```
Input join(rw)_i
Effect:
  if ¬failed then
    if status = idle then
      if i = i_0 then
        status ← active
      else
        status ← joining
      world ← world ∪ {i}

Input recv(join)_{j,i}
Effect:
  if ¬failed then
    if status ≠ idle then
      world ← world ∪ {j}

Output join-ack(rw)_i
Precondition:
  ¬failed
  status = active
Effect:
  none

Input fail_i
Effect:
  failed ← true
```

---

Figure 6: *Reader-Writer*$_i$: Join-related and failure transitions

**Information propagation.** Information is propagated between *Reader-Writer* processes in the background, via point-to-point channels that are accessed using send and recv actions. The algorithm uses only one kind of message, which contains a tuple including 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 background messages may be sent at any time, once the sender is active. They are sent only to processes in the sender's *world* set, that is, processes that the sender knows have tried to join the system at some point.

When *Reader-Writer*$_i$ receives a message, it sets its *status* to active, if it has not already done so. It adds incoming information about the world, in $W$, to its local *world* set. It compares the incoming tag $t$ to its own *tag*. It $t$ is strictly greater, it represents a more recent version of the object; in this case, *Reader-Writer*$_i$ sets its *tag* to $t$ and its *value* to the incoming value $v$. *Reader-Writer*$_i$ also updates its *cmap* with the information in the incoming configuration map, *cm*, using the *update* operator defined in Section 2. That is, for each $k$, if $cmap(k) = \bot$ and $cm(k)$ is a configuration identifier $c \in C$, then process $i$ sets its $cmap(k)$ to $c$. Also, if $cmap(k) \in C_\bot$, and $cm(k) = \pm$, indicating that the sender knows that configuration $k$ has already been garbage-collected, then *Reader-Writer*$_i$ sets its $cmap(k)$ to $\pm$. *Reader-Writer*$_i$ also updates its $pnum2(j)$ component for the sender $j$ to reflect new information about the phase number of the sender, which appears in the *pns* components of the message.

If $\mathit{Reader\text{-}Writer}_i$ is currently 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. $\mathit{Reader\text{-}Writer}_i$ uses the phase numbers to perform this check: if the incoming phase number *pnr* is at least as large as the current operation phase number (*op.pnum* or *gc.pnum*), then process $i$ knows that the message is recent. If the message is recent, then it is used to update the records for current read, write or garbage-collection operations. For more information about how this is done and why, see the descriptions of these operations below.

**Read and write operations.** A read or write operation is performed in two phases: a query phase and a propagation phase. In each phase, $\mathit{Reader\text{-}Writer}_i$ obtains recent *value*, *tag*, and *cmap* information from "enough" processes. This information is obtained by sending and receiving messages in the background, as described above.

When $\mathit{Reader\text{-}Writer}_i$ starts either a query phase or a propagation phase of a read or write, it sets *op.cmap* to a *CMap* whose configurations are intended to be used to conduct the phase. Specifically, $\mathit{Reader\text{-}Writer}_i$ chooses the *CMap* *truncate*(*cmap*), which is defined to include all the configuration identifiers in the local *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 identifiers, up to the first $\perp$ in *cm*. If adding these new configuration identifiers 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 identifiers does create a gap (that is, the result is not in *Truncated*), then $\mathit{Reader\text{-}Writer}_i$ can infer that it has been using out-of-date configuration identifiers. In this case, it restarts the phase using the best currently known *CMap*, information, which is obtained by computing *truncate*(*cmap*) for the latest local *cmap*.

In between restarts, while process $i$ is engaged in a single attempt to complete a phase, it never removes a configuration identifier from *op.cmap*, that is, the set of configuration identifiers being used for the phase is only increased. In particular, if process $i$ learns during a phase that a configuration identifier in $\mathit{op.cmap}(k)$ has been garbage-collected, it does not remove it from *op.cmap*, but continues to include it in conducting the phase.

The query phase of a read or write operation terminates when a *query fixed point* is reached. This happens when $\mathit{Reader\text{-}Writer}_i$ determines that it has received recent responses from some read-quorum of each configuration in its current *op.cmap*. Let $t$ denote process $i$'s *tag* at the query fixed point. Then we know that $t$ 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 operation, then process $i$ determines at this point that its current value is the value to be returned to its client. However, before returning this value, process $i$ embarks upon the propagation phase of the read operation, whose purpose is to make sure that "enough" *Reader-Writer* processes have acquired tags that are at least $t$ (and associated values). 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 $\mathit{Reader\text{-}Writer}_i$ has received recent responses from some write-quorum of each configuration in the current *op.cmap*. When this occurs, we know that the *tag* of each process in each of these write-quorums is at least $t$.

Processing for a write operation starting with a $\mathsf{write}(v)_i$ event is similar to that for a read

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:
  $op.phase = \mathsf{idle}$

Figure 7: $Reader\text{-}Writer_i$: Read/write transitions

operation. 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 $Reader\text{-}Writer_i$ defines the tag for its write operation to be the pair $(n+1, i)$. $Reader\text{-}Writer_i$ sets its local $tag$ to $(n+1, i)$ and its $value$ to $v$, the value it is currently writing. 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 the new tag $(n+1, i)$. The propagation phase is conducted exactly as for a read operation: Information is propagated in the background, and $op.cmap$ is managed as described above. The propagation phase is over when the same propagation fixed point condition is satisfied as for the read operation.

The communication strategy we use for reads and writes is different from what is done in other similar algorithms (e.g., [7, 18, 34]). Typically, process $i$ first determines a tag and value to propagate, and then propagates it directly to appropriate quorums. In our algorithm, communication occurs in the background, and process $i$ just checks a fixed point condition. The fixed point condition ensures that enough processes have received recent messages, which implies that they must have tags at least as large as the one that process $i$ is trying to propagate.

**New configurations and garbage collection.** When $Reader\text{-}Writer_i$ learns about a new configuration identifier via a new-config input action, it simply records it in its $cmap$. From time to time, configuration identifiers get garbage-collected at $i$, in numerical order. The configuration identifiers 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 $Reader\text{-}Writer_i$ may garbage-collect a configuration identifier, say, the one in $cmap(k)$. First, $Reader\text{-}Writer_i$ can garbage-collect $cmap(k)$ if it learns that another process has already garbage-collected it. This happens when a $\mathsf{recv}_{*,i}$ event occurs in which $cm(k) = \pm$. The second, more interesting situation is where $Reader\text{-}Writer_i$ acquires enough information to garbage-collect configuration $k$ on its own. $Reader\text{-}Writer_i$ acquires this information by carrying out a garbage-collection operation, which is a two-phase operation with a structure similar to the read and write operations. $Reader\text{-}Writer_i$ may initiate a 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 operations may proceed concurrently with read or write operations at the same node.

In the query phase of a garbage-collection operation, $Reader\text{-}Writer_i$ communicates with both a read-quorum and a write-quorum of configuration $k$. The query phase accomplishes two tasks: First, $Reader\text{-}Writer_i$ ensures that certain information is conveyed to the processes in a read-quorum and a write-quorum of $k$. In particular, all these processes learn about both configurations $k$ and $k+1$, and also learn that all configurations smaller than $k$ have been garbage-collected. We refer loosely to the fact that they know about configuration $k+1$ as the "forwarding pointer" condition— if such a process $j$, is contacted later by someone who is trying to access a quorum of configuration $k$, $j$ is able to tell that process about the existence of configuration $k+1$. Second, in the query phase, $Reader\text{-}Writer_i$ collects $tag$ and $value$ information from the read-quorum and write-quorum that it accesses. This ensures that, by the end of the query phase, $Reader\text{-}Writer_i$'s $tag$ is equal to some $t$ that is at least as great as the $tag$ that each of the quorum members had when it sent a message to $Reader\text{-}Writer_i$ for the query phase. In the propagation phase, $Reader\text{-}Writer_i$ ensures that all the processes in a write-quorum of the new configuration, $k+1$, have acquired $tags$

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 when $\mathit{Reader\text{-}Writer}_i$ is carrying out a garbage-collection operation for configuration $k$, it may discover that someone else has already garbage-collected $k$; it learns this by observing that $cmap(k) = \pm$. When this happens, $\mathit{Reader\text{-}Writer}_i$ may simply terminate its garbage-collection operation.

---

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

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

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

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

---

Figure 8: $\mathit{Reader\text{-}Writer}_i$: Garbage-collection transitions

### 5.3 Channel automata

We assume point to point channels $\mathit{Channel}_{i,j}$, one for each $i, j \in I$ (including the case where $i = j$). $\mathit{Channel}_{i,j}$ 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 $m$. We assume that message $m$ is an element of the message alphabet $M$, which we assume includes all the messages that are used by the protocol.

Channels may lose and reorder messages, but cannot manufacture new messages or duplicate messages (the latter restriction is used for convenience, in reasoning about latency). Formally, we model the channel as a multiset. A $\mathsf{send}(m)_{i,j}$ adds one copy of the message $m$ to the multiset, and a $\mathsf{recv}(m)_{i,j}$ removes one copy of $m$. A $\mathsf{lose}$ action allows any sub-multiset of messages to be lost.

### 5.4 The complete algorithm

The complete implementation $\mathcal{S}$ is the composition of all the automata defined above—the *Joiner*$_i$ and *Reader-Writer*$_i$ automata for all $i$, all the channels, and any automaton whose traces satisfy the *Recon* safety specification—with all the actions that are not external actions of RAMBO hidden.

## 6 Safety Proof

In this section, we show that our implementation $\mathcal{S}$ satisfies the safety guarantees of RAMBO, as given in Section 3, assuming the environment safety assumptions. That is, we prove the following theorem:

**Theorem 6.1** *Let $\beta$ be a trace of the system $\mathcal{S}$. If $\beta$ satisfy the* RAMBO *environment assumptions, then $\beta$ satisfies the* RAMBO *service guarantees (well-formedness and atomicity).*

This safety theorem does not depend on any assumptions about timing and failures. In contrast, our performance results, in Sections 8 and 9, do depend on such assumptions.

The proof of well-formedness is straightforward based on inspection of the code, so the rest of this section is devoted to the proof of the atomicity property. To prove atomicity, we consider a trace $\beta$ of $\mathcal{S}$ that satisfies the RAMBO environment assumptions and in which all read and write operations complete. We show the existence of a partial order on the operations in $\beta$ satisfying the conditions listed in the atomicity definition in Section 3.

The proof is carried out in several stages. First, in Section 6.1, we establish some notational conventions and define some useful history variables. In Section 6.2, we establish some simple invariants involving configuration maps. Then in Section 6.3, we present results that say what is achieved by the two phases of read, write, and garbage-collection operations. The next three subsections describe information propagation between operations: Section 6.4 describes the relationship between garbage-collection operations, Section 6.5 describes the relationship between a garbage-collection operation and a read or write operation, and Section 6.6 describes the relationship between two read or write operations. Section 6.6 culminates in Lemma 6.14, which says that tags are monotonic with respect to non-concurrent read or write operations. Finally, Section 6.7 uses the tags to define a partial order on operations and verifies the four properties required for atomicity.

Throughout Section 6, we consider executions of $\mathcal{S}$ whose traces satisfy the RAMBO environment assumptions. We call these *good* executions. In particular, an "invariant" in this section is a statement that is true of all states that are reachable in good executions of $\mathcal{S}$.

### 6.1 Notational conventions

Before diving into the proof, we introduce some notational conventions and add certain history variables to the global state of the system $\mathcal{S}$.

An *operation* can be of type read, write, or garbage-collection. Operations are uniquely identified by their starting events, that is, a read operation is defined by its $\mathsf{read}_i$ event, a write operation by its $\mathsf{write}(v)_i$ event, and a garbage-collection operation by its $\mathsf{gc}(k)_i$ event.[3]

We introduce the following history variables:

[3]An *event* is an occurrence of an action in a sequence, formally, a pair $(a, n)$, where $a$ is an action and $n$ is an index at which $a$ occurs in that sequence.

- *in-transit*, a set of messages, initially $\emptyset$.
  A message is added to the set when it is sent by any $\textit{Reader-Writer}_i$ to any $\textit{Reader-Writer}_j$. No message is ever removed from this set.

- For every $k \in \mathbb{N}$:

  - $c(k) \in C$, initially undefined.
    This is set when the first $\mathsf{new\text{-}config}(c, k)_i$ occurs, for some $c$ and $i$. It is set to the $c$ that appears as the first argument of this action.

- For every operation $\pi$:

  - $tag(\pi) \in T$, initially undefined.
    This is set to the value of *tag* at the process running $\pi$, at the point right after $\pi$'s query-fix or gc-query-fix event occurs. If $\pi$ is a read or garbage-collection operation, this is the highest tag that it encounters during the query phase. If $\pi$ is a write operation, this is the new tag that is selected for performing the write.

- For every read or write operation $\pi$:

  - $\textit{query-cmap}(\pi)$, a *CMap*, initially undefined.
    This is set in the query-fix step of $\pi$, to the value of $op.cmap$ in the pre-state.
  - $R(\pi, k)$, for $k \in \mathbb{N}$, a subset of $I$, initially undefined.
    This is set in the query-fix step of $\pi$, for each $k$ such that $\textit{query-cmap}(\pi)(k) \in C$. It is set to an arbitrary $R \in \textit{read-quorums}(c(k))$ such that $R \subseteq op.acc$ in the pre-state.
  - $\textit{prop-cmap}(\pi)$, a *CMap*, initially undefined.
    This is set in the prop-fix step of $\pi$, to the value of $op.cmap$ in the pre-state.
  - $W(\pi, k)$, for $k \in \mathbb{N}$, a subset of $I$, initially undefined.
    This is set in the prop-fix step of $\pi$, for each $k$ such that $\textit{prop-cmap}(\pi)(k) \in C$. It is set to an arbitrary $W \in \textit{write-quorums}(c(k))$ such that $W \subseteq op.acc$ in the pre-state.

- For every garbage-collection operation $\gamma$ for $k$:

  - $R(\gamma)$, a subset of $I$, initially undefined.
    This is set in the gc-query-fix step of $\gamma$, to an arbitrary $R \in \textit{read-quorums}(c(k))$, such that $R \subseteq gc.acc$ in the pre-state.
  - $W_1(\gamma)$, a subset of $I$, initially undefined.
    This is set in the gc-query-fix step of $\gamma$, to an arbitrary $W \in \textit{write-quorums}(c(k))$ such that $W \subseteq gc.acc$ in the pre-state.
  - $W_2(\gamma)$, a subset of $I$, initially undefined.
    This is set in the gc-prop-fix step of $\gamma$, to an arbitrary $W \in \textit{write-quorums}(c(k+1))$ such that $W \subseteq gc.acc$ in the pre-state.

In any good execution $\alpha$, we define the following events (more precisely, we give additional names to some existing events):

- For every read or write operation $\pi$:

- query-phase-start($\pi$), initially undefined.
  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 it is not the case that $op.acc$ is again reset to $\emptyset$ following that event and prior to the query-fix step). This is either a read, write, or recv event.
- prop-phase-start($\pi$), initially undefined.
  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.

We define a property of garbage-collection events in an execution $\alpha$:

- *Initial garbage-collection events:* A gc-prop-fix$(k)_i$ event is *initial* if it is the first gc-prop-fix$(k)_*$ event in $\alpha$. A garbage-collection operation is *initial* if its gc-prop-fix event is initial.

### 6.2 Configuration map invariants

In this subsection, we give invariants describing the kinds of configuration maps that may appear in various places in the state of $\mathcal{S}$.

The first invariant (recall this means a property of all states that arise in good executions of $\mathcal{S}$) describes some properties of $cmap_i$ that hold while $Reader\text{-}Writer_i$ is conducting a garbage-collection operation:

**Invariant 1** *If $gc.phase_i \neq$ idle and $gc.index_i = k$ then:*

1. $cmap(k)_i \in C \cup \{\pm\}$.
2. $cmap(k+1)_i \in C \cup \{\pm\}$.
3. $k = 0$ *or* $cmap(k-1) = \pm$.

**Proof.** By the precondition of gc$(k)_i$ and monotonicity of all the changes to $cmap_i$. Specifically, for this invariant, if for some $h$ we have $cmap(h) \in C$ in the pre-state, then $cmap(h) \in C \cup \{\pm\}$ in the post-state (by code inspection). □

We next proceed to describe the patterns of $C$, $\perp$, and $\pm$ values that may occur in configuration maps in various places in the system state.

**Invariant 2** *Let cm be a CMap that appears as one of the following:*

1. *The cm component of some message in in-transit.*
2. $cmap_i$ *for any* $i \in I$.
3. $op.cmap_i$ *for some* $i \in I$ *for which* $op.phase \neq$ idle.
4. *query-cmap*$(\pi)$ *or prop-cmap*$(\pi)$ *for any operation* $\pi$.

*Then* $cm \in$ *Usable.*

In the following proof and elsewhere, we use dot notation to indicate components of a state, for example, $s.cmap_i$ indicates the value of $cmap_i$ in state $s$.
**Proof.** By induction on the length of a finite good execution.

*Base:* Part 1 holds because initially, *in-transit* is empty. Part 2 holds because initially, for every $i$, $cmap(0)_i = c_0$ and $cmap(k)_i = \bot$; the resulting *CMap* is in *Usable*. Part 3 holds vacuously, because in the initial state, all *op.phase* values are idle. Part 4 also holds vacuously, because in the initial state, all *query-cmap* and *prop-cmap* variables are undefined.

*Inductive step:* Let $s$ and $s'$ be the states before and after the new event, respectively. We consider Parts 1-4 one by one.
For Part 1, the interesting case is a $\mathsf{send}_i$ event that puts a message containing $cm$ in *in-transit*. The precondition on send action implies that $cm$ is set to $s.cmap_i$. The inductive hypothesis, Part 2, implies that $s.cmap_i \in Usable$, which suffices.
For Part 2, fix $i$. The interesting cases are those that may change $cmap_i$, namely, $\mathsf{new\text{-}config}_i$, $\mathsf{recv}_i$ for a gossip (non-join) message, and $\mathsf{gc\text{-}prop\text{-}fix}_i$.

1. $\mathsf{new\text{-}config}(c, *)_i$.
   By inductive hypothesis, $s.cmap_i \in Usable$. The only change this can make is changing a $\bot$ to $c$. Then Lemma 2.2, Part 2, implies that $s'.cmap_i \in Usable$.

2. $\mathsf{recv}(\langle *, *, cm, *, * \rangle)_i$.
   By inductive hypothesis, $cm \in Usable$ and $s.cmap_i \in Usable$. The step sets $s'.cmap_i$ to $update(s.cmap_i, cm)$. Lemma 2.2, Part 1, then implies that $s'.cmap_i \in Usable$.

3. $\mathsf{gc\text{-}prop\text{-}fix}(k)_i$.
   This sets $cmap(k)_i$ to $\pm$. If $s.cmap(k)_i = \pm$, then this step causes no change and we are done. So suppose that this is not the case; then Invariant 1, Part 1, implies that $s.cmap(k)_i \in C$ and Invariant 1, Part 2, implies that $s.cmap(k{+}1)_i \in C \cup \{\pm\}$. Since $s.cmap_i \in Usable$, this implies that $s.cmap(k+1)_i \in C$. Invariant 1, Part 3, implies that either $k = 0$ or $s.cmap(k-1) = \pm$. Since $s.cmap_i \in Usable$, under the conditions just described, $s.cmap(\ell)_i = \pm$ if and only if $\ell < k$. Then changing $cmap(k)_i$ to $\pm$ preserves usability.

For Part 3, the interesting actions to consider are those that modify *op.cmap*, namely, $\mathsf{read}_i$, $\mathsf{write}_i$, $\mathsf{recv}_i$, and $\mathsf{query\text{-}fix}_i$.

1. $\mathsf{read}_i$, $\mathsf{write}_i$, or $\mathsf{query\text{-}fix}_i$.
   By inductive hypothesis, $s.cmap_i \in Usable$. The new step sets $s'.op.cmap_i$ to $truncate(s.cmap_i)$; since $s.cmap_i \in Usable$, Lemma 2.2, Part 4, implies that this is also usable.

2. $\mathsf{recv}(\langle *, *, cm, *, * \rangle)_i$.
   This step may alter $op.cmap_i$ only if $s.op.phase \in \{\mathsf{query}, \mathsf{prop}\}$, and then in only two ways: by setting it either to $extend(s.op.cmap_i, truncate(cm))$ or to $truncate(update(s.cmap_i, cm))$. The inductive hypothesis implies that $s.op.cmap_i$, $cmap_i$, and $cm$ are all in *Usable*. Lemma 2.2 implies that *truncate*, *extend*, and *update* all preserve usability. Therefore, $s'.op.cmap_i \in Usable$.

For Part 4, the actions to consider are $\mathsf{query\text{-}fix}_i$ and $\mathsf{prop\text{-}fix}_i$.

1. $\mathsf{query\text{-}fix}_i$.
   This sets $s'.query\text{-}cmap_i$ to the value of $s.op.cmap_i$. Since by inductive hypothesis that is usable, so is $s'.query\text{-}cmap_i$.

2. $\mathsf{prop\text{-}fix}_i$.
   This sets $s'.prop\text{-}cmap_i$ to the value of $s.op.cmap_i$. Since by inductive hypothesis that is usable, so is $s'.prop\text{-}cmap_i$.

$\square$

We now strengthen Invariant 2 to say more about the form of the *CMaps* that are used for read and write operations:

**Invariant 3** *Let cm be a CMap that appears as $op.cmap_i$ for some $i \in I$ for which $op.phase_i \neq \mathsf{idle}$, or as $query\text{-}cmap(\pi)$ or $prop\text{-}cmap(\pi)$ for any operation $\pi$. Then:*

1. *$cm \in Truncated$.*

2. *cm consists of finitely many $\pm$ entries followed by finitely many $C$ entries followed by an infinite number of $\perp$ entries.*

**Proof.** We prove that the desired properties hold for a *cm* that is $op.cmap_i$. The same properties for $query\text{-}cmap_i$ and $prop\text{-}cmap_i$ follow by the way they are defined, from $op.cmap_i$.

To prove Part 1 we proceed by induction. In the initial state, $op.phase_i = \mathsf{idle}$, which makes the claim vacuously true. For the inductive step we consider all actions that alter $op.cmap_i$:

1. $\mathsf{read}_i$, $\mathsf{write}_i$, or $query\text{-}fix_i$.
   These set $op.cmap_i$ to $truncate(cmap_i)$, which is necessarily in *Truncated*.

2. $\mathsf{recv}_i$.
   This first sets $op.cmap_i$ to a preliminary value and then tests if the result is in *Truncated*. If it is, we are done. If not, then this step resets $op.cmap_i$ to $truncate(cmap_i)$, which is in *Truncated*.

To see Part 2, note that $cm \in Usable$ by Invariant 2. The fact that $cm \in Truncated$ then follows from the definition of *Usable* and Part 1. $\square$

### 6.3 Phase guarantees

In this section, we present results saying what is achieved by the individual operation phases. We give four lemmas, describing the messages that must be sent and received and the information flow that must occur during the two phases of garbage-collection and during the two phases of read and write operations.

Note that these lemmas treat the case where $j = i$ uniformly with the case where $j \neq i$. This is because, in the *Reader-Writer* algorithm, communication from a location to itself is treated uniformly with communication between two different locations. We first consider the query phase of garbage-collection. Lemma 6.2 says that, in the query phase of a garbage-collection of $k$, every member $j$ of the designated read-quorum and designated write-quorum learns about all configurations up to and including the $k+1^{st}$. Moreover, the *tag* assigned to the garbage-collection operation is at least as great as the one sent by any such $j$.

**Lemma 6.2** *Suppose that a* $\mathsf{gc\text{-}query\text{-}fix}(k)_i$ *event for a garbage-collection operation* $\gamma$ *occurs in* $\alpha$. *Suppose* $j \in R(\gamma) \cup W_1(\gamma)$.
*Then there exist messages* $m$ *from* $i$ *to* $j$ *and* $m'$ *from* $j$ *to* $i$ *such that:*

1. $m$ *is sent after the* $\mathsf{gc}(k)_i$ *event of* $\gamma$.
2. $m'$ *is sent after* $j$ *receives* $m$.
3. $m'$ *is received before the* $\mathsf{gc\text{-}query\text{-}fix}(k)_i$ *event of* $\gamma$.
4. *In any state after* $j$ *receives* $m$, $cmap(\ell)_j \neq \bot$ *for all* $\ell \leq k+1$.
5. $tag(\gamma)$ *is at least as great as the value of* $tag_j$ *in any state before* $j$ *sends message* $m'$.

**Proof.** The phase number discipline implies the existence of the claimed messages $m$ and $m'$.

For Part 4, the precondition of $\mathsf{gc}(k)$ and the fact that $cmap_i \in Usable$ (by Invariant 2) together imply that, when the $\mathsf{gc}(k)_i$ event of $\gamma$ occurs, $cmap(\ell)_i \neq \bot$ for all $\ell \leq k+1$. Therefore, $j$ sets $cmap(\ell)_j \neq \bot$ for all $\ell \leq k+1$ when it receives $m$. Monotonicity of $cmap_j$ ensures that this property persists forever.

For Part 5, let $t$ be the value of $tag_j$ in any state before $j$ sends message $m'$. Let $t'$ be the value of $tag_j$ in the state just before $j$ sends $m'$, by monotonicity. Then $t \leq t'$, by monotonicity. The $tag$ component of $m'$ is equal to $t'$, by the code for $\mathsf{send}$. Since $i$ receives this message before the $\mathsf{gc\text{-}query\text{-}fix}(k)$, it follows that $tag(\gamma)$ is set by $i$ to a value $\geq t$. □

Next, we consider the propagation phase of garbage-collection. Lemma 6.3 says that, in the propagation phase of a garbage-collection, every member $j$ of the designated write-quorum acquires a *tag* that is at least as great as the *tag* of the garbage-collection operation.

**Lemma 6.3** *Suppose that a* $\mathsf{gc\text{-}prop\text{-}fix}(k)_i$ *event for a garbage-collection operation* $\gamma$ *occurs in* $\alpha$. *Suppose that* $j \in W_2(\gamma)$.
*Then there exist messages* $m$ *from* $i$ *to* $j$ *and* $m'$ *from* $j$ *to* $i$ *such that:*

1. $m$ *is sent after the* $\mathsf{gc\text{-}query\text{-}fix}(k)_i$ *event of* $\gamma$.
2. $m'$ *is sent after* $j$ *receives* $m$.
3. $m'$ *is received before the* $\mathsf{gc\text{-}prop\text{-}fix}(k)_i$ *event of* $\gamma$.
4. *In any state after* $j$ *receives* $m$, $tag_j \geq tag(\gamma)$.

**Proof.** The phase number discipline implies the existence of the claimed messages $m$ and $m'$.

For Part 4, when $j$ receives $m$, it sets $tag_j$ to be $\geq tag(\gamma)$. Monotonicity of $tag_j$ ensures that this property persists in later states. □

Next, we consider the query phase of read and write operations. Lemma 6.4 says that the *tag* assigned to a read or write operation is at least as great as the one sent in the query phase by any member $j$ of the designated read-quorum; if the operation is a write, then the *tag* is strictly greater. Also, the read or write operation learns about all configurations known by any such $j$ by the time $j$ sent its message for the query phase.

**Lemma 6.4** *Suppose that a* query-fix$_i$ *event for a read or write operation* $\pi$ *occurs in* $\alpha$. *Let* $k, k' \in \mathbb{N}$. *Suppose query-cmap*$(\pi)(k) \in C$ *and* $j \in R(\pi, k)$.
*Then there exist messages* $m$ *from* $i$ *to* $j$ *and* $m'$ *from* $j$ *to* $i$ *such that:*

1. $m$ *is sent after the* query-phase-start$(\pi)$ *event.*
2. $m'$ *is sent after* $j$ *receives* $m$.
3. $m'$ *is received before the* query-fix *event of* $\pi$.
4. *If* $t$ *is the value of* $tag_j$ *in any state before* $j$ *sends* $m'$, *then:*
   (a) $tag(\pi) \geq t$.
   (b) *If* $\pi$ *is a write operation then* $tag(\pi) > t$.
5. *If* $cmap(\ell)_j \neq \bot$ *for all* $\ell \leq k'$ *in any state before* $j$ *sends* $m'$, *then query-cmap*$(\pi)(\ell) \in C$ *for some* $\ell \geq k'$.

**Proof.** The phase number discipline implies the existence of the claimed messages $m$ and $m'$.

For Part 4, the *tag* component of message $m'$ is $\geq t$, so $i$ receives a tag that is $\geq t$ during the query phase of $\pi$. Therefore, $tag(\pi) \geq t$. Also, if $\pi$ is a write, the effects of the query-fix imply that $tag(\pi) > t$.

Finally, we show Part 5. In the $cm$ component of message $m'$, $cm(\ell) \neq \bot$ for all $\ell \leq k'$. Therefore, $truncate(cm)(\ell) = cm(\ell)$ for all $\ell \leq k'$, so $truncate(cm)(\ell) \neq \bot$ for all $\ell \leq k'$.

Let $cm'$ be the configuration map $extend(op.cmap_i, truncate(cm))$ computed by $i$ during the effects of the recv event for $m'$. Since $i$ does not reset $op.acc$ to $\emptyset$ in this step, by definition of the query-phase-start event, it follows that $cm' \in Truncated$, and $cm'$ is the value of $op.cmap_i$ just after the recv step.

Fix $\ell$, $0 \leq \ell \leq k'$. We claim that $cm'(\ell) \neq \bot$. We consider cases:

1. $op.cmap(\ell)_i \neq \bot$ just before the recv step.

   Then the definition of *extend* implies that $cm'(\ell) \neq \bot$, as needed.

2. $op.cmap(\ell)_i = \bot$ just before the recv step and $truncate(cm)(\ell) \in C$.

   Then the definition of *extend* implies that $cm'(\ell) \in C$, which implies that $cm'(\ell) \neq \bot$, as needed.

3. $op.cmap(\ell)_i = \bot$ just before the recv step and $truncate(cm)(\ell) \notin C$.

   Since $truncate(cm)(\ell) \neq \bot$, it follows that $truncate(cm)(\ell) = \pm$. Since $truncate(cm)(\ell) = \pm$ and $truncate(cm) \in Usable$, it follows that, for some $\ell' > \ell$, $truncate(cm)(\ell') \in C$.

   By the case assumption, $op.cmap(\ell)_i = \bot$ just before the recv step. Since, by Invariant 3, $op.cmap_i \in Truncated$, it follows that $op.cmap(\ell')_i = \bot$ before the recv step.

   Then by definition of *extend*, we have that $cm'(\ell) = \bot$ while $cm'(\ell') \in C$. This implies that $cm' \notin Truncated$, which contradicts the fact, already shown, that $cm' \notin Truncated$, So this case cannot arise.

Since this argument holds for all $\ell$, $0 \leq \ell \leq k'$, it follows that $cm'(\ell) \neq \perp$ for all $\ell \leq k'$. Since $cm'(\ell) \neq \perp$ for all $\ell \leq k'$, Invariant 2 implies that $cm' \in \mathit{Usable}$, which implies by definition of *Usable* that $cm'(\ell) \in C$ for some $\ell \geq k'$. That is, $op.cmap_i(\ell) \in C$ for some $\ell \geq k'$ immediately after the recv step. This implies that $\mathit{query\text{-}cmap}(\pi)(\ell) \in C$ for some $\ell \geq k'$, as needed. □

Finally, we consider the propagation phase of read and write operations. Lemma 6.5 says that, in the propagation phase of a read or write, every member $j$ of the designated write-quorum acquires a *tag* that is at least as great as the *tag* of the read or write operation. Also, the read or write operation learns about all configurations known by any such $j$ by the time $j$ sent its message for the propagation phase.

**Lemma 6.5** *Suppose that a* prop-fix$_i$ *event for a read or write operation* $\pi$ *occurs in* $\alpha$. *Suppose* $\mathit{prop\text{-}cmap}(\pi)(k) \in C$ *and* $j \in W(\pi, k)$.
*Then there exist messages* $m$ *from* $i$ *to* $j$ *and* $m'$ *from* $j$ *to* $i$ *such that:*

1. $m$ *is sent after the* prop-phase-start$(\pi)$ *event.*

2. $m'$ *is sent after* $j$ *receives* $m$.

3. $m'$ *is received before the* prop-fix *event of* $\pi$.

4. *In any state after* $j$ *receives* $m$, $tag_j \geq tag(\pi)$.

5. *If* $cmap(\ell)_j \neq \perp$ *for all* $\ell \leq k'$ *in any state before* $j$ *sends* $m'$, *then* $\mathit{prop\text{-}cmap}(\pi)(\ell) \in C$ *for some* $\ell \geq k'$.

**Proof.** The phase number discipline implies the existence of the claimed messages $m$ and $m'$.

For Part 4, let $m.tag$ be the *tag* field of message $m$. Since $m$ is sent after the prop-phase-start event, which is not earlier than the query-fix, it must be that $m.tag \geq tag(\pi)$. Therefore, by the effects of the recv, just after $j$ receives $m$, $tag_j \geq m.tag \geq tag(\pi)$. Then monotonicity of $tag_j$ implies that $tag_j \geq tag(\pi)$ in any state after $j$ receives $m$.

For Part 5, the proof is analogous to the proof of Part 5 of Lemma 6.4. In fact, it is identical except for the final conclusion, which now says that $\mathit{prop\text{-}cmap}(\pi)(\ell) \in C$ for some $\ell \geq k'$. □

### 6.4 Behavior of garbage-collection

In this subsection, we present lemmas describing information flow between garbage-collection operations. The first lemma says that initial gc-prop-fix$(k)$ events for successive $k$ occur in order. In fact, for each $k$, the initial gc-prop-fix$(k)$ event precedes any attempt by any process to garbage-collect $k + 1$. This means that garbage-collection obeys a simple, sequential discipline.

**Lemma 6.6** 1. *If any* gc$(\ell)_i$ *event occurs in* $\alpha$ *and* $0 \leq k < \ell$, *then some* gc-prop-fix$(k)$ *event occurs in* $\alpha$, *and the initial* gc-prop-fix$(k)$ *event precedes the given* gc$(\ell)_i$ *event.*

2. *If any* gc-prop-fix$(\ell)$ *event occurs in* $\alpha$ *and* $0 \leq k < \ell$, *then some* gc-prop-fix$(k)$ *event occurs in* $\alpha$, *and the initial* gc-prop-fix$(k)$ *event precedes the given* gc-prop-fix$(\ell)$ *event.*

**Proof.** For Part 1, note that the precondition of $\mathsf{gc}(\ell)_i$ implies that $cmap(\ell-1)_i = \pm$ in the pre-state. Since $cmap_i \in Usable$, it must be that $cmap(k)_i = \pm$ for all $k$, $0 \le k < \ell$, in the pre-state. This implies that prior to the $\mathsf{gc}(\ell)_i$, some $\mathsf{gc\text{-}prop\text{-}fix}(k)$ event occurs in $\alpha$ for each $k$, $0 \le k < \ell$.

Part 2 follows from Part 1 and the behavior of garbage-collection operations. □

The sequential nature of garbage-collection has a nice consequence for propagation of tags: Consider a particular good execution $\alpha$. For any $k \in \mathbb{N}$, define $\gamma_k$ to be the initial garbage-collection operation for $k$, if any $\mathsf{gc\text{-}prop\text{-}fix}(k)$ event occurs. If no such event occurs, then $\gamma_k$ is undefined. The lemma says that the *tag*s of garbage-collection operations are monotonically nondecreasing with respect to the configuration indices.

**Lemma 6.7** *Suppose a* $\mathsf{gc\text{-}query\text{-}fix}(\ell)$ *event for* $\gamma_\ell$ *occurs in* $\alpha$ *and* $k \le \ell$. *Then* $tag(\gamma_k) \le tag(\gamma_\ell)$.

**Proof.** Fix $k$. We use induction on $\ell$.
The base case, $\ell = k$, is trivially true. For the inductive step, assume that $\ell \ge k+1$ and the result is true for $\ell - 1$. To show that the result is true for $\ell$, assume that a $\mathsf{gc\text{-}query\text{-}fix}(\ell)$ event for $\gamma_\ell$ occurs in $\alpha$. Then Lemma 6.6 implies that $\mathsf{gc\text{-}query\text{-}fix}(\ell-1)$ for $\gamma_{\ell-1}$ also occurs in $\alpha$. Therefore, by inductive hypothesis, $tag(\gamma_k) \le tag(\gamma_{\ell-1})$. It suffices to show that $tag(\gamma_{\ell-1}) \le tag(\gamma_\ell)$.

By Lemma 6.6, the $\mathsf{gc\text{-}prop\text{-}fix}(\ell-1)$ for $\gamma_{\ell-1}$ occurs in $\alpha$ and precedes the $\mathsf{gc}(\ell)$ event of $\gamma_\ell$. Then $R(\gamma_\ell)$ and $W_2(\gamma_{\ell-1})$ are both defined in $\alpha$. Since both are quorums of $c(\ell)$, they have a nonempty intersection; choose $j \in R(\gamma_\ell) \cap W_2(\gamma_{\ell-1})$.

Lemma 6.3 and monotonicity of $tag_j$ imply that, in any state after the $\mathsf{gc\text{-}prop\text{-}fix}(\ell-1)$ for $\gamma_{\ell-1}$, $tag_j \ge tag(\gamma_{\ell-1})$. Lemma 6.6 implies that this $\mathsf{gc\text{-}prop\text{-}fix}(\ell-1)$ precedes the $\mathsf{gc}(\ell)$ event of $\gamma_\ell$. Therefore, $t \ge tag(\gamma_{\ell-1})$, where $t$ is defined to be the value of $tag_j$ just before the $\mathsf{gc}(\ell)$ event of $\gamma_\ell$. Lemma 6.2 and monotonicity of $tag_j$ imply that $tag(\gamma_\ell) \ge t$. Thus, we have $tag(\gamma_{\ell-1}) \le t \le tag(\gamma_\ell)$, so $tag(\gamma_{\ell-1}) \le tag(\gamma_\ell)$, as needed. □

## 6.5 Behavior of a read or a write following a garbage-collection

Now we describe the relationship between a garbage-collection and a following read or write operation. The first two lemmas describe situations in which certain configurations must belong to the *query-cmap* of a read or write operation.

First, if no garbage-collection operation for $k$ completes before the $\mathsf{query\text{-}phase\text{-}start}$ event of a read or write operation, then some configuration with index $\le k$ must be included in the *query-cmap*.

**Lemma 6.8** *Let* $\pi$ *be a read or write operation whose* $\mathsf{query\text{-}fix}$ *event occurs in* $\alpha$. *Suppose that no* $\mathsf{gc\text{-}prop\text{-}fix}(k)$ *event precedes the* $\mathsf{query\text{-}phase\text{-}start}(\pi)$ *event.*
*Then* $query\text{-}cmap(\pi)(\ell) \in C$ *for some* $\ell \le k$.

**Proof.** Since no garbage-collection operation for $k$ completes before the $\mathsf{query\text{-}phase\text{-}start}(\pi)$ event, it follows that, just after the $\mathsf{query\text{-}phase\text{-}start}(\pi)$ event, $op.cmap(k) \ne \pm$; that is, $op.cmap(k) \in C_\perp$. Then Invariant 2 implies that, just after $\mathsf{query\text{-}phase\text{-}start}(\pi)$, $op.cmap(\ell) \in C$ for some $\ell \le k$. Fix such an $\ell$; then the behavior of the query phase of the read or write implies that $query\text{-}cmap(\pi)(\ell) \in C$. □

Second, if some garbage-collection for $k$ does complete before the $\mathsf{query\text{-}phase\text{-}start}$ event of a read or write operation, then some configuration with index $\ge k+1$ must be included in the *query-cmap*.

**Lemma 6.9** *Let $\gamma$ be a 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 $query\text{-}cmap(\pi)(\ell) \in C$ for some $\ell \geq k+1$.*

**Proof.** Suppose for the sake of contradiction that $query\text{-}cmap(\pi)(\ell) \notin C$ for all $\ell \geq k+1$. Fix $k' = \max(\{\ell : query\text{-}cmap(\pi)(\ell) \in C\})$. Then $k' \leq k$. Since the gc-prop-fix$(k)$ event of $\gamma$ precedes the query-phase-start$(\pi)$ event, Lemma 6.6 implies that a gc-prop-fix$(k')$ event also precedes the query-phase-start$(\pi)$ event. Let $\gamma'$ be the initial garbage-collection operation for $k'$.

Then write-quorum $W_1(\gamma')$ of $c(k')$ and read-quorum $R(\pi, k')$ are both defined; choose $j \in W_1(\gamma') \cap R(\pi, k')$. Then Lemma 6.2 and monotonicity of *cmap* imply that, in the state just prior to the gc-prop-fix$(k')$ event of $\gamma'$, $cmap(\ell)_j \neq \bot$ for all $\ell \leq k'+1$. Then Lemma 6.4 implies that $query\text{-}cmap(\pi)(\ell) \in C$ for some $\ell \geq k'+1$. But this contradicts the choice of $k'$. $\square$

The next two lemmas describe propagation of *tag* information from a garbage-collection operation to a following read or write operation. The first lemma assumes that the *query-cmap* of the read or write includes the configuration following the one being garbage-collected.

**Lemma 6.10** *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. Suppose also that $query\text{-}cmap(\pi)(k+1) \in C$. Then:*

1. *$tag(\gamma) \leq tag(\pi)$.*

2. *If $\pi$ is a write operation then $tag(\gamma) < tag(\pi)$.*

**Proof.** The propagation phase of $\gamma$ accesses write-quorum $W_2(\gamma)$ of $c(k+1)$, whereas the query phase of $\pi$ accesses read-quorum $R(\pi, k+1)$. Since both are quorums of configuration $c(k+1)$, they have a nonempty intersection; choose $j \in W_2(\gamma) \cap R(\pi, k+1)$.

Lemma 6.3 implies that, in any state after the gc-prop-fix$(k)$ event for $\gamma$, $tag_j \geq tag(\gamma)$. Since the gc-prop-fix$(k)$ event of $\gamma$ precedes the query-phase-start$(\pi)$ event, we have that $t \geq tag(\gamma)$, where $t$ is defined to be the value of $tag_j$ just before the query-phase-start$(\pi)$ event. Then Lemma 6.4 implies that $tag(\pi) \geq t$, and if $\pi$ is a write operation, then $tag(\pi) > t$. Combining the inequalities yields both conclusions of the lemma. $\square$

The final lemma has a similar statement to the previous one. However, this one drops the assumption that the *query-cmap* of the read or write includes the configuration following the one being garbage-collected.

**Lemma 6.11** *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:*

1. *$tag(\gamma) \leq tag(\pi)$.*

2. *If $\pi$ is a write operation then $tag(\gamma) < tag(\pi)$.*

**Proof.** Lemma 6.9 implies that $query\text{-}cmap(\pi)(\ell) \in C$ for some $\ell \geq k+1$.
Let $k' = \min(\{\ell : query\text{-}cmap(\pi)(\ell) \in C\})$. We consider cases:

1. $k' \leq k + 1$.

   Then Invariant 3 implies that *query-cmap*$(\pi)(k+1) \in C$, that is, that configuration $k+1$ is included in the query phase of $\pi$. Then Lemma 6.10 implies the conclusions.

2. $k' > k + 1$.

   By Lemma 6.8, some gc-prop-fix$(k' - 1)$ event precedes the query-phase-start$(\pi)$ event. Let $\gamma'$ be the initial garbage-collection operation for $k' - 1$; then the gc-prop-fix$(k' - 1)$ event of $\gamma'$ precedes the query-phase-start$(\pi)$ event. Since $k < k' - 1$, Lemma 6.7 implies that $tag(\gamma) \leq tag(\gamma')$.

   Since *query-cmap*$(\pi)(k') \in C$, we may apply Lemma 6.10 to $\gamma'$ and $\pi$, which yields that $tag(\gamma') \leq tag(\pi)$, and if $\pi$ is a write, then $tag(\gamma') < tag(\pi)$. Combining the inequalities yields both conclusions of the lemma.

□

## 6.6 Behavior of sequential reads and writes

Read or write operations that originate at different locations may proceed concurrently. However, in the special case where they execute sequentially, we can prove some relationships between their *query-cmap*s, *prop-cmap*s, and *tag*s. The first lemma says that, when two read or write operations execute sequentially, the smallest configuration index used in the propagation of the first operation is less than or equal to the largest index used in the query phase of the second. In other words, we cannot have a situation in which the second operation's query phase executes using only configurations with indices that are strictly less than any used in the first operation's propagation phase.

**Lemma 6.12** *Assume $\pi_1$ and $\pi_2$ are two read or write operations, such that:*

1. *The* prop-fix *event of $\pi_1$ occurs in $\alpha$.*

2. *The* query-fix *event of $\pi_2$ occurs in $\alpha$.*

3. *The* prop-fix *event of $\pi_1$ precedes the* query-phase-start$(\pi_2)$ *event.*

*Then* $\min(\{\ell : \textit{prop-cmap}(\pi_1)(\ell) \in C\}) \leq \max(\{\ell : \textit{query-cmap}(\pi_2)(\ell) \in C\})$.

**Proof.** Suppose for the sake of contradiction that $\min(\{\ell : \textit{prop-cmap}(\pi_1)(\ell) \in C\}) > k$, where $k$ is defined to be $\max(\{\ell : \textit{query-cmap}(\pi_2)(\ell) \in C\})$. Then in particular, *prop-cmap*$(\pi_1)(k) \notin C$. The form of *prop-cmap*$(\pi_1)$, as expressed in Invariant 3, implies that *prop-cmap*$(\pi_1)(k) = \pm$.

This implies that some gc-prop-fix$(k)$ event occurs prior to the prop-fix of $\pi_1$, and hence prior to the query-phase-start$(\pi_2)$ event. Lemma 6.9 then implies that *query-cmap*$(\pi_2)(\ell) \in C$ for some $\ell \geq k + 1$. But this contradicts the choice of $k$. □

The next lemma describes propagation of *tag* information, in the case where the propagation phase of the first operation and the query phase of the second operation share a configuration.

**Lemma 6.13** *Assume $\pi_1$ and $\pi_2$ are two read or write operations, and $k \in \mathbb{N}$, such that:*

1. *The* prop-fix *event of* $\pi_1$ *occurs in* $\alpha$.

2. *The* query-fix *event of* $\pi_2$ *occurs in* $\alpha$.

3. *The* prop-fix *event of* $\pi_1$ *precedes the* query-phase-start$(\pi_2)$ *event.*

4. *prop-cmap*$(\pi_1)(k)$ *and query-cmap*$(\pi_2)(k)$ *are both in* $C$.

*Then:*

1. $tag(\pi_1) \leq tag(\pi_2)$.

2. *If* $\pi_2$ *is a write then* $tag(\pi_1) < tag(\pi_2)$.

**Proof.** The hypotheses imply that $\textit{prop-cmap}(\pi_1)(k) = \textit{query-cmap}(\pi_2)(k) = c(k)$. Then $W(\pi_1, k)$ and $R(\pi_2, k)$ are both defined in $\alpha$. Since they are both quorums of configuration $c(k)$, they have a nonempty intersection; choose $j \in W(\pi_1, k) \cap R(\pi_2, k)$.

Lemma 6.5 implies that, in any state after the prop-fix event of $\pi_1$, $tag_j \geq tag(\pi_1)$. Since the prop-fix event of $\pi_1$ precedes the query-phase-start$(\pi_2)$ event, we have that $t \geq tag(\pi_1)$, where $t$ is defined to be the value of $tag_j$ just before the query-phase-start$(\pi_2)$ event. Then Lemma 6.4 implies that $tag(\pi_2) \geq t$, and if $\pi_2$ is a write operation, then $tag(\pi_2) > t$. Combining the inequalities yields both conclusions. $\square$

The final lemma is similar to the previous one, but it does not assume that the propagation phase of the first operation and the query phase of the second operation share a configuration. The main focus of the proof is on the situation where all the configuration indices used in the query phase of the second operation are greater than those used in the propagation phase of the first operation.

**Lemma 6.14** *Assume* $\pi_1$ *and* $\pi_2$ *are two read or write operations, such that:*

1. *The* prop-fix *of* $\pi_1$ *occurs in* $\alpha$.

2. *The* query-fix *of* $\pi_2$ *occurs in* $\alpha$.

3. *The* prop-fix *event of* $\pi_1$ *precedes the* query-phase-start$(\pi_2)$ *event.*

*Then:*

1. $tag(\pi_1) \leq tag(\pi_2)$.

2. *If* $\pi_2$ *is a write then* $tag(\pi_1) < tag(\pi_2)$.

**Proof.** Let $i_1$ and $i_2$ be the indices of the processes that run operations $\pi_1$ and $\pi_2$, respectively. Let $cm_1 = \textit{prop-cmap}(\pi_1)$ and $cm_2 = \textit{query-cmap}(\pi_2)$. If there exists $k$ such that $cm_1(k) \in C$ and $cm_2(k) \in C$, then Lemma 6.13 implies the conclusions of the lemma. So from now on, we assume that no such $k$ exists.

Lemma 6.12 implies that $\min(\{\ell : cm_1(\ell) \in C\}) \leq \max(\{\ell : cm_2(\ell) \in C\})$. Invariant 3 implies that the set of indices used in each phase consists of consecutive integers. Since the intervals have no indices in common, it follows that $k_1 < k_2$, where $k_1$ is defined to be $\max(\{\ell : cm_1(\ell) \in C\})$ and $k_2$ is defined to be $\min(\{\ell : cm_2(\ell) \in C\})$.

Since, for every $k \leq k_2 - 1$, $query.cmap(\pi_2)(k) \notin C$, Lemma 6.8 implies that, for every $k \leq k_2 - 1$, a $\mathsf{gc\text{-}prop\text{-}fix}(k)$ event occurs before the $\mathsf{query\text{-}phase\text{-}start}(\pi_2)$ event. For each such $k$, that is, for $0 \leq k \leq k_2 - 1$, define $\gamma_k$ to be the initial garbage-collection operation for $k$.

We focus now on the relationship between $\pi_1$ and $\gamma_{k_1}$. The propagation phase of $\pi_1$ accesses write-quorum $W(\pi_1, k_1)$ of configuration $c(k_1)$, whereas the query phase of $\gamma_{k_1}$ accesses read-quorum $R(\gamma_{k_1})$ of configuration $k_1$. Since $W(\pi_1, k_1) \cap R(\gamma_{k_1}) \neq \emptyset$, we may fix some $j \in W(\pi_1, k_1) \cap R(\gamma_{k_1})$. Let message $m$ from $i_1$ to $j$ and message $m'$ from $j$ to $i_1$ be as in Lemma 6.5. Let message $m_1$ from the process running $\gamma_{k_1}$ to $j$ and message $m_1'$ from $j$ to the process running $\gamma_{k_1}$ be the messages whose existence is asserted in Lemma 6.2.

We claim that $j$ sends $m'$, its message for $\pi_1$, before it sends $m_1'$, its message for $\gamma_{k_1}$. Suppose for the sake of contradiction that $j$ sends $m_1'$ before it sends $m'$. Lemma 6.2 implies that, just before $j$ sends $m_1'$, $cmap(k)_j \neq \perp$ for all $k \leq k_1 + 1$. Since $j$ sends $m_1'$ before it sends $m'$, monotonicity of $cmap$ implies that just before $j$ sends $m'$, $cmap(k)_j \neq \perp$ for all $k \leq k_1 + 1$. Then Lemma 6.5 implies that $prop\text{-}cmap(\pi_1)(\ell) \in C$ for some $\ell \geq k_1 + 1$. But this contradicts the choice of $k_1$, which implies that $j$ sends $m'$ before it sends $m_1'$.

Since $j$ sends $m'$ before it sends $m_1'$, Lemma 6.5 implies that, at the time $j$ sends $m_1'$, $tag(\pi_1) \leq tag_j$. Then Lemma 6.2 implies that $tag(\pi_1) \leq tag(\gamma_{k_1})$.

Since $k_1 \leq k_2 - 1$, Lemma 6.7 implies that $tag(\gamma_{k_1}) \leq tag(\gamma_{k_2-1})$. Lemma 6.11 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 the various inequalities then yields both conclusions. □

## 6.7 Atomicity

Let $\beta$ be a trace of $\mathcal{S}$ that satisfies the RAMBO environment assumptions, and assume that all read and write operations complete in $\beta$. Consider any particular (good) execution $\alpha$ of $\mathcal{S}$ whose trace is $\beta$.[4] We define a partial order $\prec$ on read and write operations in $\beta$, in terms of the operations' tags in $\alpha$. Namely, we totally order the writes in order of their *tag*s, and we order each read with respect to all the writes as follows: a read with $tag = t$ is ordered after all writes with $tag \leq t$ and before all writes with $tag > t$.

**Lemma 6.15** *The ordering $\prec$ is well-defined.*

**Proof.** The key is to show that no two write operations get assigned the same tag. This is obviously true for two writes that are initiated at different locations, because the low-order tiebreaker identifiers are different. For two writes at the same location, Lemma 6.14 implies that the tag of the second is greater than the tag of the first. This suffices. □

**Lemma 6.16** *$\prec$ satisfies the four conditions in the definition of atomicity.*[5]

**Proof.** We begin with Condition 2, which (as usual in such proofs), is the most interesting thing to show. Suppose for the sake of contradiction that $\pi_1$ completes before $\pi_2$ starts, yet $\pi_2 \prec \pi_1$. We consider two cases:

[4] The "scope" of these definitions of $\alpha$ and $\beta$ is just the following two lemmas and their proofs.
[5] The four conditions for atomicity are defined in Section 3.

($i$) $\pi_2$ is a write operation.
Since $\pi_1$ completes before $\pi_2$ starts, Lemma 6.14 implies that $tag(\pi_2) > tag(\pi_1)$. On the other hand, the fact that $\pi_2 \prec \pi_1$ implies that $tag(\pi_2) \leq tag(\pi_1)$. This yields a contradiction.

($ii$) $\pi_2$ is a read operation.
Since $\pi_1$ completes before $\pi_2$ starts, Lemma 6.14 implies that $tag(\pi_2) \geq tag(\pi_1)$. On the other hand, the fact that $\pi_2 \prec \pi_1$ implies that $tag(\pi_2) < tag(\pi_1)$. This yields a contradiction.

Since we have a contradiction in either case, Property 2 must hold.

Condition 1 follows from Condition 2 with the following observation. Consider any operation $\pi$ in an execution where all the read and write operations complete. Given that $\pi$ terminates, any operation that starts after it terminates cannot be ordered before $\pi$, by Condition 2. Since only a finite number of operations can start before the termination of $\pi$, then only a subset of such operations can be ordered before $\pi$.

Conditions 3 and 4 are straightforward. □

Now we tie everything together for the proof of Theorem 6.1.

**Proof.** (of Theorem 6.1)
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 the atomicity condition (as defined in Section 3), assume that all read and write operations complete in $\beta$. Let $\alpha$ be a good execution of $\mathcal{S}$ whose trace is $\beta$. Define the ordering $\prec$ on the read and write operations in $\beta$ as above, using the chosen $\alpha$. Then Lemma 6.16 says that $\prec$ satisfies the four conditions in the definition of atomicity. Thus, $\beta$ satisfies the atomicity condition, as needed. □

# 7 Implementation of the Reconfiguration Service

In this section, we describe a distributed algorithm that implements the *Recon* service. We also describe how to combine this algorithm with the components already defined in Section 5, thus obtaining the complete RAMBO system.

We describe the implementation of *Recon* for a particular object $x$ (and we suppress mention of $x$). The *Recon* algorithm 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 proposed new configurations. The configuration that $Cons(k, c)$ decides upon 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 $Reader\text{-}Writer_i$ 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.

### 7.1 Consensus services

In this subsection, we specify the behavior we assume for consensus service $Cons(k,c)$, for a fixed $k \geq 1$ and $c \in C$. Fix $V$ to be the set of consensus values. (In the implementation of the *Recon* service, $V$ will be instantiated as $C$, the set of configuration identifiers.) The external signature of $Cons(k,c)$ is given in Figure 9.

| Input: | Output: |
|---|---|
| $\mathsf{init}(v)_{k,c,i}$, $v \in V$, $i \in members(c)$ | $\mathsf{decide}(v)_{k,c,i}$, $v \in V$, $i \in members(c)$ |
| $\mathsf{fail}_i$, $i \in members(c)$ | |

Figure 9: $Cons(k,c)$: External signature

We describe the safety properties of $Cons(k,c)$ in terms of properties of a trace $\beta$ of actions in the external signature. Namely, we define the environment safety assumptions:

- *Well-formedness:* For any $i \in members(c)$:
  - No $\mathsf{init}(*)_{k,c,i}$ event is preceded by a $\mathsf{fail}_i$ event.
  - At most one $\mathsf{init}(*)_{k,c,i}$ event occurs in $\beta$.

And we define the consensus safety guarantees:

- *Well-formedness:* For any $i \in members(c)$:
  - No $\mathsf{decide}(*)_{k,c,i}$ event is preceded by a $\mathsf{fail}_i$ event.
  - At most one $\mathsf{decide}(*)_{k,c,i}$ event occurs in $\beta$.
  - If a $\mathsf{decide}(*)_{k,c,i}$ event occurs in $\beta$, then it is preceded by an $\mathsf{init}(*)_{k,c,i}$ event.
- *Agreement:* If $\mathsf{decide}(v)_{k,c,i}$ and $\mathsf{decide}(v')_{k,c,i'}$ events occur in $\beta$, then $v = v'$.
- *Validity:* If a $\mathsf{decide}(v)_{k,c,i}$ event occurs in $\beta$, then it is preceded by an $\mathsf{init}(v)_{k,c,j}$.

The behavior specified above can be achieved using the Paxos consensus algorithm [30], as described formally in [14]. We call this version of the Paxos algorithm $\textsc{Paxos}_{impl}$. $\textsc{Paxos}_{impl}$ uses a fixed parameter $\varepsilon > 0$; in the rest of this section, we fix $\varepsilon$.

The following theorem says that $\textsc{Paxos}_{impl}$ satisfies the safety guarantees described above, based on the safety assumptions:

**Theorem 7.1** *If $\beta$ is a trace of $\textsc{Paxos}_{impl}$ that satisfies the safety assumptions of $Cons(k,c)$, then $\beta$ also satisfies the (well-formedness, agreement, and validity) safety guarantees of $Cons(k,c)$.*

$\textsc{Paxos}_{impl}$ also satisfies the following latency result [14]:

**Theorem 7.2** *Consider a timed execution $\alpha$ of $\textsc{Paxos}_{impl}$ and a prefix $\alpha'$ of $\alpha$. Suppose that:*

1. *The underlying system "behaves well" after $\alpha'$, in the sense that timing is "normal" (what is called "regular" in [14])[6] and no process failures or message losses occur.*

[6] In [14], regular timing implies that messages are delivered within time $d$, that local processing time is 0, and that information is "gossiped" at intervals of $d$.

2. *For every $i$ that does not fail in $\alpha$, an* $\mathsf{init}(*)_i$ *event occurs in $\alpha'$.*

3. *There exist $R \in$ read-quorums$(c)$ and $W \in$ write-quorums$(c)$ such that for all $i \in R \cup W$, no* $\mathsf{fail}_i$ *event occurs in $\alpha$.*

*Then for every $i$ that does not fail in $\alpha$, a* $\mathsf{decide}(*)_i$ *event occurs, no later than $10d + \varepsilon$ time after the end of $\alpha'$.*

In our latency analysis, in Sections 8 and 9, we assume that the $Cons(k, c)$ services are implemented using $\text{PAXOS}_{impl}$.

## 7.2 Recon automata

The signature and state of $Recon_i$ appear in Figures 10 and the transitions in Figure 11.

**Signature:**

Input:
$\mathsf{join}(\mathsf{recon})_i$
$\mathsf{recon}(c, c')_i$, $c, c' \in C, i \in members(c)$
$\mathsf{decide}(c)_{k,i}$, $c \in C, k \in \mathbb{N}^+$
$\mathsf{recv}(\langle \mathsf{config}, c, k\rangle)_{j,i}$, $c \in C$, $k \in \mathbb{N}^+$, $i \in members(c)$, $j \in I - \{i\}$
$\mathsf{recv}(\langle \mathsf{init}, c, c', k\rangle)_{j,i}$, $c, c' \in C$, $k \in \mathbb{N}^+$, $i, j \in members(c)$, $j \neq i$
$\mathsf{fail}_i$

Output:
$\mathsf{join\text{-}ack}(\mathsf{recon})_i$
$\mathsf{new\text{-}config}(c, k)_i$, $c \in C, k \in \mathbb{N}^+$
$\mathsf{init}(c, c')_{k,i}$, $c, c' \in C, k \in \mathbb{N}^+$, $i \in members(c)$
$\mathsf{recon\text{-}ack}(b)_i$, $b \in \{\mathsf{ok}, \mathsf{nok}\}$
$\mathsf{report}(c)_i$, $c \in C$
$\mathsf{send}(\langle \mathsf{config}, c, k\rangle)_{i,j}$, $c \in C$, $k \in \mathbb{N}^+$, $j \in members(c) - \{i\}$
$\mathsf{send}(\langle \mathsf{init}, c, c', k\rangle)_{i,j}$, $c, c' \in C, k \in \mathbb{N}^+$, $i, j \in members(c)$, $j \neq i$

**State:**

*status* $\in \{\mathsf{idle}, \mathsf{active}\}$, initially $\mathsf{idle}$.
*rec-cmap* $\in$ *CMap*, initially *rec-cmap*$(0) = c_0$ and *rec-cmap*$(k) = \bot$ for all $k \neq 0$.
*did-new-config* $\subseteq \mathbb{N}^+$, initially $\emptyset$
*reported* $\subseteq C$, initially $\emptyset$

*op-status* $\in \{\mathsf{idle}, \mathsf{active}\}$, initially $\mathsf{idle}$
*op-outcome* $\in \{\mathsf{ok}, \mathsf{nok}, \bot\}$, initially $\bot$
*cons-data* $\in (\mathbb{N}^+ \to (C \times C))$, initially everywhere $\bot$
*did-init* $\subseteq \mathbb{N}^+$, initially $\emptyset$
*failed*, a Boolean, initially *false*

Figure 10: $Recon_i$: Signature and state

$Recon_i$ begins operating by setting its *status* variable to $\mathsf{active}$, when a $\mathsf{join}(\mathsf{recon})$ input event occurs. $Recon_i$ responds to such a $\mathsf{join}$ input with a $\mathsf{join\text{-}ack}(\mathsf{recon})_i$ output event.

$Recon_i$'s state includes a variable *rec-cmap*, which holds a *CMap*: *rec-cmap*$(k) = c$ indicates that $i$ knows that $c$ is the $k^{th}$ configuration identifier. If $Recon_i$ has learned that $c$ is the $k$th configuration identifier, it can convey this information to its local $Reader\text{-}Writer_i$ using a $\mathsf{new\text{-}config}(c, k)_i$ output action; variable *did-new-config* keeps track of the indices for which $Recon_i$ has done a $\mathsf{new\text{-}config}$ output. $Recon_i$ can also convey the fact that $c$ is the $k^{th}$ configuration identifier to other $Recon_j$, $j \in members(c)$, using a $\langle \mathsf{config}, c, k\rangle$ message. Also, $Recon_i$ can inform its local client that $c$ is the latest configuration identifier that it knows about, using a $\mathsf{report}(c)_i$ output action.

$Recon_i$ learns about a configuration identifier in one of two ways: either directly, by receiving a $\mathsf{decide}$ input from a *Cons* service, or indirectly, by receiving a $\mathsf{config}$ or $\mathsf{init}$ message from another $Recon_j$ automaton.

$Recon_i$ receives a reconfiguration request from its environment via a $\mathsf{recon}(c, c')_i$ event, where $i \in members(c)$. (An environment well-formedness assumption says that the environment waits for

Input $\mathsf{join}(\mathsf{recon})_i$
Effect:
  if $\neg failed$ then
    if $status = \mathsf{idle}$ then
      $status \leftarrow \mathsf{active}$

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

Output $\mathsf{new\text{-}config}(c, k)_i$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $rec\text{-}cmap(k) = c$
  $k \notin did\text{-}new\text{-}config$
Effect:
  $did\text{-}new\text{-}config \leftarrow did\text{-}new\text{-}config \cup \{k\}$

Output $\mathsf{send}(\langle \mathsf{config}, c, k\rangle)_{i,j}$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $rec\text{-}cmap(k) = c$
Effect:
  none

Input $\mathsf{recv}(\langle \mathsf{config}, c, k\rangle)_{j,i}$
Effect:
  if $\neg failed$ then
    if $status = \mathsf{active}$ then
      $rec\text{-}cmap(k) \leftarrow c$

Output $\mathsf{report}(c)_i$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $c = rec\text{-}cmap(k)$
  $\forall \ell > k : rec\text{-}cmap(\ell) = \perp$
  $c \notin reported$
Effect:
  $reported \leftarrow reported \cup \{c\}$

Input $\mathsf{recon}(c, c')_i$
Effect:
  if $\neg failed$ then
    if $status = \mathsf{active}$ then
      $op\text{-}status \leftarrow \mathsf{active}$
      let $k = \max(\{\ell : rec\text{-}cmap(\ell) \in C\})$
      if $c = rec\text{-}cmap(k)$ and $cons\text{-}data(k+1) = \perp$ then
        $cons\text{-}data(k+1) \leftarrow \langle c, c'\rangle$
        $op\text{-}outcome \leftarrow \perp$
      else
        $op\text{-}outcome \leftarrow \mathsf{nok}$

Output $\mathsf{init}(c')_{k,c,i}$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $cons\text{-}data(k) = \langle c, c'\rangle$
  if $k \geq 1$ then $k - 1 \in did\text{-}new\text{-}config$
  $k \notin did\text{-}init$
Effect:
  $did\text{-}init \leftarrow did\text{-}init \cup \{k\}$

Output $\mathsf{send}(\langle \mathsf{init}, c, c', k\rangle)_{i,j}$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $cons\text{-}data(k) = \langle c, c'\rangle$
  $k \in did\text{-}init$
Effect:
  none

Input $\mathsf{recv}(\langle \mathsf{init}, c, c', k\rangle)_{j,i}$
Effect:
  if $\neg failed$ then
    if $status = \mathsf{active}$ then
      if $rec\text{-}cmap(k-1) = \perp$ then $rec\text{-}cmap(k-1) \leftarrow c$
      if $cons\text{-}data(k) = \perp$ then $cons\text{-}data(k) \leftarrow \langle c, c'\rangle$

Input $\mathsf{decide}(c')_{k,c,i}$
Effect:
  if $\neg failed$ then
    if $status = \mathsf{active}$ then
      $rec\text{-}cmap(k) \leftarrow c'$
      if $op\text{-}status = \mathsf{active}$ then
        if $cons\text{-}data(k) = \langle c, c'\rangle$ then $op\text{-}outcome \leftarrow \mathsf{ok}$
        else $op\text{-}outcome \leftarrow \mathsf{nok}$

Output $\mathsf{recon\text{-}ack}(b)_i$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $op\text{-}status = \mathsf{active}$
  $op\text{-}outcome = b$
Effect:
  $op\text{-}status = \mathsf{idle}$

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



Figure 11: $Recon_i$: Transitions.

any previous reconfiguration request at the same location to complete (with a recon-ack) before issuing another request.) Upon receiving such a request, if $c$ is the latest configuration identifier $Recon_i$ knows about, $Recon_i$ prepares data for participating in consensus on the configuration identifier to follow $c$. This data is a pair consisting of $c$, the latest known configuration identifier, and $c'$, the proposed new configuration identifier. If $c$ is not the latest configuration, $Recon_i$ prepares to respond negatively to the new reconfiguration request, by setting *op-outcome* to nok. Variable *op-status* records the existence of a locally-initiated reconfiguration request, and variable *op-outcome* is used to keep track of the planned response value.

$Recon_i$ can initiate participation in a $Cons(k, c)$ algorithm, with an $\mathsf{init}(*)_{k,c,i}$ output event, after its consensus data are prepared. Before doing so, it makes sure that it has already notified $Reader\text{-}Writer_i$ about the current configuration $c$. Variable *did-init* keeps track of the values of $k$ for which $i$ has initiated participation in some $Cons(k, *)$ service; this is used to prevent $i$ from participating in consensus for the same $k$ more than once. After initiating participation in a consensus algorithm, $Recon_i$ sends init messages to inform the other members of the current configuration $c$ about its initiation of consensus. Another member who receives this information may use it to prepare to participate in the same consensus algorithm. Such a member may also take advantage of the received information to include the current configuration in its *rec-cmap*. Thus, there are two ways in which $Recon_i$ can initiate participation in consensus: as a result of a local recon event, or by receiving an init message from another $Recon_j$ process.

When $Recon_i$ receives a $\mathsf{decide}(c')_{k,c,i}$ directly from $Cons(k, c)$, it records configuration $c'$ as the $k^{th}$ configuration identifier in its *rec-cmap*. It also determines whether a response to its local client is necessary (if a local reconfiguration operation is active), and determines the response based on whether the consensus decision is the same as the locally-proposed configuration identifier. $Recon_i$ actively informs members of $c'$ that $c'$ is the $k^{th}$ configuration, by sending config messages. It does not notify anyone else. The consensus service $Cons(k, c)$ is responsible for conveying consensus decisions to $members(c)$.

**Theorem 7.3** *Let $\beta$ be a trace of the Recon implementation. If $\beta$ satisfies the Recon environment assumptions, then $\beta$ satisfies the Recon service guarantees (well-formedness, agreement, validity, and no duplication).*

### 7.3 The complete RAMBO system

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 using $\text{PAXOS}_{impl}$. We use the same kinds of channels as for RAMBO: point-to-point channels, one for each $i, j \in I$; again, the channels may lose and reorder messages, but may not manufacture new messages or duplicate messages.

The complete RAMBO system (for a particular object) consists of *Joiner*, *Reader-Writer*, and *Channel* automata as described in Section 5, plus $Recon_{impl}$. We denote the complete RAMBO system by $\mathcal{S}'$.

We finish this section by defining two properties of configuration indices in an execution $\alpha$ of $\mathcal{S}'$. Let $k \in \mathbb{N}$. Then:

- *Latest configuration index:* Index $k$ is the *latest* configuration index in $\alpha$ provided that one of the following holds:

1. $k = 0$ and no $\mathsf{decide}$ event occurs in $\alpha$.
2. A $\mathsf{decide}(*)_{k,*,*}$ event occurs in $\alpha$ and no $\mathsf{decide}(*)_{k+1,*,*}$ event occurs in $\alpha$.

We say that $c \in C$ is the *latest* configuration identifier in $\alpha$ provided that one of the following holds:

1. $c = c_0$ and 0 is the latest configuration index in $\alpha$.
2. A $\mathsf{decide}(c)_{k,*,*}$ event occurs in $\alpha$ and $k$ is the latest configuration index.

- *Installed configuration index:* Index $k$ is *installed* in $\alpha$ provided that either $k = 0$ or there exists $c \in C$ such that both of the following hold:

  1. At least one $\mathsf{init}(*)_{k,c,*}$ event occurs in $\alpha$.
  2. For every $i \in \mathit{members}(c)$, either a $\mathsf{decide}(*)_{k,c,i}$ event or a $\mathsf{fail}_i$ event occurs in $\alpha$.

  That is, the $k-1^{st}$ configuration is $c$, and every non-failed member of $c$ has learned about the $k^{th}$ configuration. If index $k$ is installed and $\mathit{rec\text{-}cmap}(k)_i = c'$ for some $i$ and $c'$, then we also say that configuration $c'$ is *installed*.

# 8 Latency Bounds: Normal Behavior Throughout the Execution

In this section and Section 9, we present our conditional performance results—latency results for the various operations performed by RAMBO under various assumptions about timing, failures, and the patterns of requests. This section contains results for executions in which "normal" timing and failure behavior occurs throughout the execution, whereas Section 9 contains results for executions in which normal behavior occurs from some point onward. We formulate these results for the full RAMBO system $\mathcal{S}'$ consisting of $\mathit{Reader\text{-}Writer}_i$ and $\mathit{Joiner}_i$ for all $i$, $\mathit{Recon}_{impl}$ (which consists of $\mathit{Recon}_i$ for all $i$ and $\mathit{Cons}(k,c)$ for all $k$ and $c$), and channels between all $i$ and $j$. Since we are dealing here with timing, we "convert" all these automata to general timed automata as defined in [33], by allowing arbitrary amounts of time to pass in any state, without changing the state.

Section 8.1 describes restrictions on the nondeterministic choices within the RAMBO algorithm, in particular, on the scheduling of locally controlled actions. We impose these restrictions for the rest of this paper. Section 8.2 describes the restrictions on timing and failure patterns that define the normal timing and failure behavior considered in this section. Section 8.3 contains some basic definitions and assumptions that are used in stating hypotheses for particular conditional performance results in this section. Section 8.4 contains latency results that do not depend on background gossiping, but only on communication that is triggered naturally by the operations. Finally, Section 8.5 contains latency results that do depend on gossiping.

## 8.1 Restricting nondeterminism

RAMBO in its full generality is a highly nondeterministic algorithm. For example, it allows sending of gossip messages at arbitrary times. In this section and Section 9, we restrict RAMBO's nondeterminism so that messages are sent at the earliest possible time and at regular intervals thereafter, and so that non-send locally controlled events occur just once, as soon as they are enabled.

More precisely, fix $d > 0$, the *normal message delay*, and fix $\varepsilon > 0$ to be the value of $\varepsilon$ used in $\mathrm{PAXOS}_{impl}$, as we described in Section 7.1. We assume a restricted version of RAMBO in which each $Joiner_i$, $Reader\text{-}Writer_i$, and $Recon_i$ automaton has a real-valued local clock, which evolves according to a continuous, monotone increasing function from nonnegative reals to reals. Local clocks of different automata (even different automata at the same location) may run at different rates. Moreover, the following conditions hold, in all admissible timed executions (those timed executions in which the limit time is $\infty$):

- *Periodic gossip:* Each $Joiner_i$ whose $status = \mathsf{joining}$ sends join messages to everyone in its *hints* set, every time $d$, according to its local clock. Each $Reader\text{-}Writer_i$ sends messages to everyone in its *world* every time $d$, according to its clock. Each $Recon_i$ sends config messages and init messages to every process to whom it is allowed to send such messages, every time $d$, according to its clock.

- *Important Joiner messages:* Each $Joiner_i$ sends a join message immediately to location $j$, without any time passing on its local clock, in the following situation:

  – Just after a $\mathsf{join}(\mathsf{rambo}, J)$ event, if $j \in J$.

- *Important Reader-Writer messages:* Each $Reader\text{-}Writer_i$ sends a message immediately to location $j$, without any time passing on its clock, in each of the following situations:

  – Just after a $\mathsf{recv}(\mathsf{join})_{j,i}$ event, if $status_i = \mathsf{active}$.
  This is when $i$ learns that $j$ is attempting to join.

  – Just after a $\mathsf{recv}(*, *, *, *, pns, *)_{j,i}$ event occurs, if $pns > pnum2(j)_i$ and $status_i = \mathsf{active}$.
  This is when $i$ receives a message from $j$ that indicates that $j$ is engaged in a later operation phase than $i$ previously knew about.

  – Just after a $\mathsf{new\text{-}config}(c, k)_i$ event, if $status_i = \mathsf{active}$ and $j \in world_i$.
  This is when $i$ learns about a new configuration from *Recon*, and $j$ is in $i$'s current *world*.

  – Just after a $\mathsf{read}_i$, $\mathsf{write}_i$, or $\mathsf{query\text{-}fix}_i$ event, or a recv event that resets $op.acc$ to $\emptyset$, if $j \in members(c)$, for some $c$ that appears in the new $op.cmap_i$.
  This is when $i$ starts or restarts a phase and $j$ is a member of a relevant configuration.

  – Just after a $\mathsf{gc}(k)_i$ event, if $j \in members(cmap(k)_i)$.
  This is when $i$ starts garbage-collecting a configuration that includes $j$ as a member.

  – Just after a $\mathsf{gc\text{-}query\text{-}fix}(k)_i$ event, if $j \in members(cmap(k+1)_i)$.
  This is when $i$ starts the second phase of the garbage-collection of a configuration, and $j$ is member of the next configuration.

- *Important Recon messages:* Each $Recon_i$ sends a message immediately to $j$, without any time passing on its clock, in the following situations:

  – The message is of the form $(\mathsf{config}, c, k)$, a $\mathsf{decide}(c)_{k,*,i}$ event has just occurred, and $j \in members(c) - \{i\}$.
  This is when $i$ has learned directly from the consensus service, about configuration $k$ and $j$ is a member of that configuration.

  – The message is of the form $(\mathsf{init}, c, c', k)$, an $\mathsf{init}(c')_{k,c,i}$ event has just occurred, and $j \in members(c) - \{i\}$.
  This is when $i$ has just initiated consensus and $j$ is another member of the configuration that is involved in performing the consensus.

- *Non-communication events:* Any non-send locally controlled action of any RAMBO automaton that has no effect on the state is performed only once, and before any time passes on the local clock.

An alternative to listing all these properties is to add appropriate bookkeeping to the various RAMBO automata to ensure these properties. This approach would help in detecting and avoiding ambiguities in the statements of the constraints. However, it would add complexity to the code. So we postpone this for now.

## 8.2 Normal behavior

The previous subsection described restrictions on the nondeterministic choices made by the algorithm. Our results also require restrictions on timing and failure behavior—things that are not generally considered to be under the control of the algorithm. Thus, we define "normal" executions as follows:

- *Normal execution*: An admissible timed execution $\alpha$ is *normal* if it satisfies the following conditions:

  1. *Regular timing behavior for* RAMBO *automata:* The local clocks of all $Joiner_i$, $Reader\text{-}Writer_i$, and $Recon_i$ automata progress at exactly the rate of real time, throughout $\alpha$.
     Recall from Section 8.1 that the timing of gossip messages, of sending events for important messages, and of the performance of other locally-controlled events, are all governed by the local clocks. Thus, this single assumption, that the local clocks progress at the rate of real time, implies that the timing of all locally-controlled events observes real-time constraints.
  2. *Reliable message delivery:* No message sent in $\alpha$ is lost.
  3. *Message delay bound:* Every message that is received in $\alpha$ is received within time $d$ of when it was sent.
  4. *Normal timing for consensus:* Timing for all consensus services is "normal".[7]

Many of our results also require assumptions about certain processes not failing for certain intervals of time. However, since these assumptions are different for different results, we postpone stating such assumptions until they are needed.

## 8.3 Hypotheses for latency results

In this section, we list various hypotheses that we need for our latency bound results. These hypotheses are needed in addition to the restrictions on nondeterminism described in Section 8.1 and the normal behavior assumptions described in Section 8.2.

The first hypothesis we define says that, when a client proposes a configuration $c$, every member of configuration $c$ must have already joined the system, at least time $e$ ago. The requirement that each member has already joined the system is already included in the environment assumptions for the RAMBO and *Recon* services; this new hypothesis adds a timing requirement:

[7] What this means internally to the consensus services is defined in [14]. As noted in Section 7.1, it means that messages are delivered within time $d$, that local processing time is 0, and that information is gossiped at intervals of $d$.

- *Reconfiguration-readiness:* Let $\alpha$ be a timed execution, $e \in \mathbb{R}^{\geq 0}$. Then $\alpha$ satisfies *e-recon-readiness* provided that, if a $\mathsf{recon}(*, c)_i$ event occurs at time $t$ then for every $j \in \mathit{members}(c)$, the event $\mathsf{join\text{-}ack}(\mathsf{rambo})_j$ occurs by time $t - e$.

The next hypothesis states a bound on the time for two participants that join the system to learn about each other.

- *Join-connectivity:* Let $\alpha$ be an admissible timed execution, $e \in \mathbb{R}^{\geq 0}$. We say $\alpha$ satisfies *e-join-connectivity* provided that, if $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ and $\mathsf{join\text{-}ack}(\mathsf{rambo})_j$ both occur in $\alpha$ by time $t$, and neither $i$ nor $j$ fails by time $t + e$, then by time $t + e$, $i \in \mathit{world}_j$.

We do not think of join-connectivity as a primitive assumption. Rather, it is a property one might expect to prove of all executions that satisfy some more basic assumptions, such as sufficient spacing between join requests. Since there are many possibilities here, we postpone considering this, and use join-connectivity itself as an assumption.

The next hypothesis, *configuration-viability*, is a reliability property for quorums. In general, in systems that use quorum configurations, operations that use quorums are guaranteed to terminate only if certain quorums do not fail. In this paper, our termination guarantees for reconfiguration, garbage-collection, and read and write operations all require assumptions that say that some quorums do not fail. Because our algorithm uses different configurations at different times, our notion of configuration-viability hypothesis takes into account which configurations might still be in use.

- *Configuration-viability:* Let $\alpha$ be an admissible timed execution, $e \in \mathbb{R}^{\geq 0}$. Then we say that $\alpha$ is *e-configuration-viable* provided that the following holds: For every $c$ and $k$ such that some $\mathit{rec\text{-}cmap}(k)_* = c$ in some state in $\alpha$, there exist $R \in \mathit{read\text{-}quorums}(c)$ and $W \in \mathit{write\text{-}quorums}(c)$ such that at least one of the following holds:

  1. No process in $R \cup W$ fails in $\alpha$.
  2. There exists a finite prefix $\alpha'$ of $\alpha$ such that $k + 1$ is installed in $\alpha'$ and no process in $R \cup W$ fails in $\alpha$ by time $\ell\mathit{time}(\alpha') + e$.

  (For a finite timed execution $\alpha$, we define $\ell\mathit{time}(\alpha)$, the *limit time* of $\alpha$, to be the time of the last event in $\alpha$.)

Note that the special case of 0-configuration-viability is not a completely trivial property. It says that certain processes remain alive until a time that is strictly greater than the time when configuration $k + 1$ is installed. This implies that events that are required to happen within 0 time of this installation must actually happen, if time subsequently increases.

The $e$-configuration-viability property is useful only in situations where a configuration is no longer needed for performing operations after time $e$ after the next configuration is installed. This latter condition holds, for suitable $e$, in RAMBO executions in which certain timing assumptions hold; the strength of those timing assumptions determines the value of $e$ that must be considered. Roughly speaking, $e$ should be sufficiently large to allow information about a new configuration to be propagated to all the active participants and for the previous configuration to be garbage-collected.

We believe that the $e$-configuration-viability assumption is reasonable for a reconfigurable algorithm such as RAMBO. This is because the algorithm can be reconfigured when quorums appear to be in danger of failing. New configurations should be chosen to minimize the likelihood of failure.

In some situations, we will not be able to characterize interesting executions in terms of *e-configuration-viability* for a fixed $e$ because an arbitrary amount of time may elapse from when a configuration becomes installed until it is garbage-collected. Therefore we define executions in which no quorum system is ever disabled:

- $\infty$-*configuration-viability*: Let $\alpha$ be an admissible timed execution. Then we say that $\alpha$ is $\infty$-*configuration-viable* provided that the following holds: For every $c$ and $k$ such that some $\mathit{rec\text{-}cmap}(k)_* = c$ in some state in $\alpha$, there exist $R \in \mathit{read\text{-}quorums}(c)$ and $W \in \mathit{write\text{-}quorums}(c)$ such that no process in $R \cup W$ fails in $\alpha$.

The next property says that a reconfiguration request waits at least a certain amount of time after a corresponding report event. Recall that environment assumptions for RAMBO and *Recon* say that such a report event must precede the request; the new assumption says that it must have occurred sufficiently long ago.

- *Recon-spacing:* Let $\alpha$ be an admissible timed execution, $e \in \mathbb{R}^{\geq 0}$. We say that $\alpha$ satisfies *e-recon-spacing* provided that, for any $\mathsf{recon}(c, *)_i$ that occurs in $\alpha$, the time since the corresponding $\mathsf{report}(c)_i$ event is $\geq e$.

Finally, the following property says that infinitely many configurations are produced. This is simply a technical assumption that is used to simplify some of our results.

- *Infinite reconfiguration:* Let $\alpha$ be an admissible timed execution. We say that $\alpha$ satisfies *infinite reconfiguration* provided for every $k \in \mathbb{N}^+$, $\alpha$ contains a $\mathsf{decide}(*)_{k,*,*}$ event.

## 8.4 Bounds that do not depend on gossiping

We give bounds for joining, reconfiguration, and garbage-collection operations for normal admissible executions. We also give bounds on reading and writing in "stable" situations. These bounds do not depend on the periodic gossiping among the $\mathit{Reader\text{-}Writer}_i$ components.

### 8.4.1 Joining

The following result gives bounds on the time to join. The result has two parts, based on whether or not the joiner is the creator of the object. Namely, if $\mathsf{join}(\mathsf{rambo}, J)_i$ occurs and $i$ does not fail, then: (1) if $i = i_0$ then the join is acknowledged immediately (within zero time), and (2) if $i \neq i_0$, $j \in J$, $\mathsf{join\text{-}ack}(\mathsf{rambo})_j$ occurs before the join of $i$, and $j$ does not fail, then $j$'s join is acknowledged within $2d$ time. More formally:

**Theorem 8.1** *Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$. If $\mathsf{join}(\mathsf{rambo}, J)_i$ occurs in $\alpha$ and $\mathsf{fail}_i$ does not occur then:*

1. *If $i = i_0$ then $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ occurs before any time elapses.*
2. *Suppose that $i \neq i_0$. Suppose also that, for some $j \in J - \{i\}$, a $\mathsf{join\text{-}ack}(\mathsf{rambo})_j$ event occurs prior to the $\mathsf{join}(\mathsf{rambo}, J)_i$ event, and $\mathsf{fail}_j$ does not occur. Then $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ occurs within time $2d$ of the $\mathsf{join}(\mathsf{rambo}, J)_i$ event.*

**Proof.** Part 1 immediately follows from the code for the creator automata, $Joiner_{i_0})$, $Reader\text{-}Writer_{i_0}$ and $Recon_{i_0}$. This is because the response does not depend on the receipt of any messages. Part 2 follows from the fact that at most two message delays are incurred by the protocol, and from the guarantee that process $j$ responds. $\square$

### 8.4.2 Reconfiguration

The next result gives a latency bound for reconfiguration, assuming no relevant failures and assuming viability. It says that, in a 0-configuration-viable execution, if $\mathsf{recon}(c,*)_i$ occurs at time $t$ and no process in $members(c)$ fails after this event, then the $\mathsf{recon}(c,*)_i$ is acknowledged with a $\mathsf{recon\text{-}ack}(*)_i$ no later than time $t+11d+\varepsilon$. More formally:

**Theorem 8.2** *Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$ satisfying 0-configuration-viability, and $t \in \mathbb{R}^{\geq 0}$. Assume that:*

1. *A $\mathsf{recon}(c,c')_i$ event occurs at time $t$ in $\alpha$.*
2. *No $\mathsf{fail}$ event for a member of $c$ occurs in $\alpha$ after the $\mathsf{recon}(c,c')_i$ event.*

*Then a $\mathsf{recon\text{-}ack}(*)_i$ event matching the assumed $\mathsf{recon}(c,c')_i$ event occurs by time $t+11d+\varepsilon$.*

**Proof.** We know that $i$, the originator of the operation, does not fail, because the signature restrictions for *Recon* require that $i \in members(c)$, and assumption 2 says that no members of $c$ fail after the $\mathsf{recon}(c,c')_i$ event.

When $\mathsf{recon}(c,c')_i$ occurs, if *outcome* is immediately set to $\mathsf{nok}$, then the time until the $\mathsf{recon\text{-}ack}(\mathsf{nok})_i$ is 0. If not, then process $i$ sets $cons\text{-}data_i$ in preparation for consensus, again within 0 time. Then an $\mathsf{init}(c')_{k,c,i}$ event occurs for some $k$ within 0 time. Then we claim that, after no more than time $10d+\varepsilon$, a $\mathsf{decide}(*)_{k,c,i}$ occurs, and before any further time passes, $\mathsf{recon\text{-}ack}(*)_i$ occurs.

The argument that $\mathsf{decide}(*)_{k,c,i}$ occurs within time $10d+\varepsilon$ proceeds as follows: First, the last $\mathsf{init}$ event for $Cons(k,c)$ that occurs in $\alpha$ must occur within time $d$ after the $\mathsf{init}(c')_{k,c,i}$ event. This is guaranteed by the sending of $\mathsf{init}$ messages by the $Recon_i$ component.

Let $\alpha'$ be the shortest prefix of $\alpha$ that includes all the $\mathsf{init}(*)_{k,c,*}$ events that occur in $\alpha$. We will apply Theorem 7.2 to $\alpha$ and $\alpha'$, to conclude that by $10d+\varepsilon$ time after the end of $\alpha'$, and hence by time $t+11d+\varepsilon$, a $\mathsf{decide}_{k,c,j}$ occurs for every non-failed $j \in members(c)$. Since $\mathsf{recon\text{-}ack}$ events happen within time 0 of the $\mathsf{decide}$ events, this will yield the result.

Applying Theorem 7.2 requires some care: we must show that the three hypotheses of that theorem are satisfied. For Property 1, the "normal case" assumptions of this section imply that timing is regular and no message losses occur after $\alpha'$. No process failures occur either: since the $\mathsf{init}(c')_{k,c,i}$ event follows the $\mathsf{recon}(c,c')_i$ event, an assumption of this theorem implies that no $\mathsf{fail}$ events for members of $c$ occur in $\alpha$ after $\alpha'$.

To see Property 2, note that the environment well-formedness conditions for RAMBO imply that all members of $c'$ must have already joined the RAMBO system when the $\mathsf{recon}(c,c')_i$ event occurs. Then they are ready to accept the $\mathsf{init}$ messages when they receive them from $i$, and they perform $\mathsf{init}(*)_{k,c,*}$ events, provided they have not failed. Since all the $\mathsf{init}(*)_{k,c,i}$ events that occur in $\alpha$ actually occur in $\alpha'$, this implies Property 2.

Property 3 is slightly tricky: it says formally that some read-quorum and some write-quorum of $c$ must stay non-failed forever. However, because of the on-line nature of the computation, Theorem 7.2 does not need that the members of these quorums stay alive after the time of the last decide event. The fact that they remain non-failed for this long follows directly from the 0-*configuration-viability* assumption. $\square$

The next result describes a situation in which the system is guaranteed to produce a positive response to a reconfiguration request.

**Theorem 8.3** *Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$, and $c \in C$. Suppose that some* $\mathsf{recon}(c,*)_*$ *event occurs in $\alpha$. Then for some $i$ such that a* $\mathsf{recon}(c,*)_i$ *event occurs in $\alpha$, either $\alpha$ contains no matching* $\mathsf{recon\text{-}ack}(b)_i$ *or* $b = \mathsf{ok}$.

**Proof.** Environment well-formedness assumptions imply that for every $i$ such that $\mathsf{recon}(c,*)_i$ occurs, there is a preceding $\mathsf{report}(c)_i$, whose precondition states that there exists $k$ such that $c = \mathit{rec\text{-}cmap}(k)_i$. By the no-duplication property of *Recon*, this must be the same $k$ for all $i$. Therefore, all the $\mathsf{recon}(c,*)_i$ events result in participation in the same consensus service, $\mathit{Cons}(k+1,c)$.

Validity of $\mathit{Cons}(k+1,c)$ implies that the decision is a configuration submitted by one of the participating members, say $i$. Then the only possible response at $i$ is $\mathsf{recon\text{-}ack}(\mathsf{ok})_i$. $\square$

### 8.4.3 Garbage-collection

The next result gives a latency bound for garbage-collection, assuming that none of the relevant processes fail. Suppose a garbage-collection operation starts with a $\mathsf{gc}(k)_i$ event. If there exist a read-quorum and a write-quorum of configuration $k$ and a write-quorum of configuration $k+1$ such that no processes in these quorums fail, and if $i$ itself does not fail, then garbage-collection terminates with $\mathsf{gc\text{-}ack}(k)_i$ within time $4d$. Formally:

**Theorem 8.4** *Let $\gamma$ be a garbage-collection operation in a normal admissible timed execution of $\mathcal{S}'$. Let $\gamma$ start with* $\mathsf{gc}(k)_i$ *and let $c_k$ and $c_{k+1}$ be the values of $\mathit{cmap}(k)_i$ and $\mathit{cmap}(k+1)_i$ when $\gamma$ starts.*
*Let $R \in \mathit{read\text{-}quorums}(c_k)$, $W_1 \in \mathit{write\text{-}quorums}(c_k)$, $W_2 \in \mathit{write\text{-}quorums}(c_{k+1})$. Assume:*

1. *Process $i$ does not fail.*

2. *No process in $R \cup W_1 \cup W_2$ fails.*

*Then $\gamma$ ends with a* $\mathsf{gc\text{-}ack}(k)_i$, *within time $4d$ of the* $\mathsf{gc}(k)_i$.

**Proof.** Since $i$ does not fail, the existence of non-failing quorums $R$, $W_1$ and $W_2$ ensures that $i$ receives replies as needed in the two phases of garbage collection. Each phase takes at most $2d$ time. $\square$

### 8.4.4 Reads and writes

The following theorem gives a bound for read and write operations in the simple "quiescent" situation where all joins and configuration management events stop from some point onward.

**Theorem 8.5** *(Informally stated.) Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$, and $\alpha'$ be a finite prefix of $\alpha$. Suppose that:*

1. *$\alpha$ is 0-viable.*
2. *The system is "quiescent" after $\alpha'$, in the sense that:*
   (a) *There are no pending join, garbage-collection, or recon requests, and no active consensus executions at the end of $\alpha'$.*
   (b) *No new* join *or* recon *requests occur in $\alpha$ after $\alpha'$.*
   (c) *Every process that has ever performed a* join-ack *and has not failed is "up-to-date" after $\alpha'$, in that its cmap consists of exactly one configuration index, which is the latest configuration, preceded by $\pm$ entries and followed by $\perp$ entries.*
3. *A* $\mathsf{read}_i$ *or* $\mathsf{write}_i$ *is initiated in $\alpha$ after $\alpha'$.*

*Then the time until a matching* $\mathsf{read\text{-}ack}_i$ *or* $\mathsf{write\text{-}ack}_i$ *event is at most $4d$.*

The next theorem describes another situation in which a read or write operation is guaranteed to have latency at most $4d$: when no new configurations are being generated, and the configuration map of the operation's initiator includes the latest configuration. This configuration map may contain more than one configuration. Since the configurations are used concurrently by the read or write operation, the use of multiple configurations does not slow the operation down. Here, we need to assume $\infty$-*configuration-viability*.

**Theorem 8.6** *Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$ satisfying $\infty$-configuration-viability, and $t \in \mathbb{R}^{\geq 0}$. Suppose $\alpha$ contains no* decide *events after time $t$, and let $k$ be the latest configuration index in $\alpha$. If a read or write operation starts in a state where $cmap(\ell)_i \neq \perp$ for all $\ell$, $0 \leq \ell \leq k$, then it completes in at most $4d$ time.*

**Proof.** This result follows from the two-phased implementation of read and write operations. Each phase lasts for at most two message delays: since new configurations are not added to $op.cmap_i$ during the phase, the phase completes in $2d$ time. New configurations can only be added in the effects of the recv action in *Reader-Writer*$_i$. Because $k$ is the latest configuration index, no higher numbered configurations exist, and smaller numbered configurations cannot be added because of the properties of the *extend* and *truncate* functions used to modify $op.cmap_i$ in the effects of recv. □

## 8.5 Bounds that depend on gossiping

In this subsection, we give results that depend on periodic gossiping among the *Reader-Writer*$_i$ automata. These results give bounds for learning about new configurations. They also give bounds on garbage-collection, and describe conditions under which garbage-collection is guaranteed to keep up with reconfiguration. Finally, we give bounds on the latency of read and write operations.

For this entire subsection, we fix $e \in \mathbb{R}^{\geq 0}$. Also, for a timed execution $\alpha$, we let $time(\pi)$ stand for the real time at which the event $\pi$ occurs in $\alpha$.

### 8.5.1 Joining

The following lemma says that if $\mathsf{report}(c)_*$ occurs then all members of $c$ are "old enough", that is, they have joined at least time $e$ earlier.

**Lemma 8.7** *Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$ satisfying e-recon-readiness, and $c \in C$, $c \neq c_0$, $i \in I$. Suppose that a $\mathsf{report}(c)_*$ event occurs at time $t$ in $\alpha$ and $i \in members(c)$. Then a $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ event occurs by time $t - e$.*

**Proof.** Assume that $\alpha$, $c$, and $i$ are as given, and that $rec\text{-}cmap(k)_i = c$ when the $\mathsf{report}(c)_i$ event occurs, that is, $c$ is the $k^{th}$ configuration. Since $c \neq c_0$, we have that $k \geq 1$. Therefore, the $\mathsf{report}(c)_*$ event is preceded by a $\mathsf{recon}(c', c)_*$ event. Then recon-readiness implies that a $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ event occurs at a time at least $e$ before the $recon(c', c)_*$ event, and so, by time $t - e$, as needed. $\square$

The next lemma says that a process receiving a $\mathsf{report}$ must be "old enough", that is, they have joined at least time $e$ earlier.

**Lemma 8.8** *Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$ satisfying e-recon-readiness, and $c \in C$, $c \neq c_0$, $i \in I$. Suppose that a $\mathsf{report}(c)_i$ event occurs at time $t$ in $\alpha$. Then a $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ event occurs by time $t - e$.*

**Proof.** Assume that $\alpha$, $c$, and $i$ are as given, and that $rec\text{-}cmap(k)_i = c$ when the $\mathsf{report}(c)_i$ event occurs, that is, $c$ is the $k^{th}$ configuration. Since $c \neq c_0$, we have that $k \geq 1$. The behavior of $Recon_i$ implies that $i$ is a member either of $c$ or of the $k - 1^{st}$ configuration, say $c'$. If $i \in members(c)$ then Lemma 8.7 implies the conclusion. So in the rest of the proof, assume that $i \in members(c')$.

If $k = 1$, then $c' = c_0$, so $i = i_0$, which implies that a $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ event occurs prior to any other $\mathsf{join\text{-}ack}(\mathsf{rambo})_*$ event. In particular, a $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ event occurs at a time that is less than or equal to that of any $\mathsf{join\text{-}ack}(\mathsf{rambo})_*$ event for any member of $c$. Since such $\mathsf{join\text{-}ack}$ events occur by time $\leq t - e$, again by Lemma 8.7, the $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ also occurs by time $t - e$, as needed.

The only other possibility is that $i \in members(c')$ and $k \geq 2$. In this case, the $\mathsf{report}(c)_i$ event must be preceded by a $\mathsf{recon}(*, c')_*$ event. Then recon-readiness implies that a $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ event occurs at a time at least $e$ before the $\mathsf{recon}(*, c')_*$ event, and so again, by time $t - e$. $\square$

### 8.5.2 Learning about configurations

The following result says that all participants succeed in exchanging information about configurations, within a short time. If both $i$ and $j$ are "old enough" (have joined at least time $e$ ago), and don't fail, then any information that $i$ has about configurations is conveyed to $j$ within time $2d$.

**Lemma 8.9** *Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$ satisfying e-join-connectivity, $t \in \mathbb{R}^{\geq 0}$, $t \geq e$. Suppose:*

1. $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ *and* $\mathsf{join\text{-}ack}(\mathsf{rambo})_j$ *both occur in $\alpha$ by time $t - e$.*
2. *Process $i$ does not fail by time $t + d$ and $j$ does not fail by time $t + 2d$.*

*Then the following hold:*

1. *If by time $t$, $cmap(k)_i \neq \perp$, then by time $t + 2d$, $cmap(k)_j \neq \perp$.*

2. *If by time $t$, $cmap(k)_i = \pm$, then by time $t + 2d$, $cmap(k)_j = \pm$.*

**Proof.** Since $\alpha$ is $e$-join-connected, by time $t$, $j \in world_i$. Sometime strictly after time $t$ and no later than time $t + d$, $Reader\text{-}Writer_i$ sends a gossip message to $j$, and $Reader\text{-}Writer_j$ receives this message by time $t + 2d$. To see Part 1, suppose that $cmap(k)_i \neq \perp$ by time $t$. Then the gossip message has $cm(k) \neq \perp$. The receipt of this message causes $j$ to set $cmap(k)_j$ to be non-$\perp$ (if it isn't already), as needed. To see Part 2, suppose that $cmap(k)_i = \pm$ by time $t$. Then the gossip message has $cm(k) = \pm$. The receipt of this message causes $j$ to set $cmap(k)_j$ to be $\pm$ (if it isn't already), as needed. □

Next, we show that, if a $\mathsf{report}(c)_i$ event occurs and $i$ does not fail, then another process $j$ learns about $c$ soon after the later of the $\mathsf{report}$ event and the time of $j$'s joining.

**Theorem 8.10** *Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$ satisfying e-recon-readiness and e-join-connectivity, $c \in C$, $k \in \mathbb{N}$, $i, j, \in I$, $t, t' \in \mathbb{R}^{\geq 0}$. Suppose:*

1. *A $\mathsf{report}(c)_i$ occurs at time $t$ in $\alpha$, where $c = rec\text{-}cmap(k)_i$, and $i$ does not fail by $\max(t, t') + d$.*

2. *$\mathsf{join\text{-}ack(rambo)}_j$ occurs in $\alpha$ by time $t' - e$, and $j$ does not fail by time $\max(t, t') + 2d$.*

*Then by time $\max(t, t') + 2d$, $cmap(k)_j \neq \perp$.*

**Proof.** The case where $k = 0$ is trivial to prove, because everyone's $cmap(0)$ is always non-$\perp$. So assume that $k \geq 1$.

Lemma 8.8 implies that $\mathsf{join\text{-}ack(rambo)}_i$ occurs by time $t - e \leq \max(t, t') - e$. Also, $\mathsf{join\text{-}ack(rambo)}_j$ occurs by time $t' - e \leq \max(t, t') - e$. By assumption, $i$ does not fail by time $\max(t, t') + d$. and $j$ does not fail by time $\max(t, t') + 2d$. Furthermore, we claim that, by time $\max(t, t')$, $cmap(k)_i \neq \perp$. This is because the time of the $report(c)_i$ is $\leq \max(t, t')$, when the $\mathsf{report}(c)_i$ occurs, $rec\text{-}cmap(k)_i \neq \perp$, and within 0 time, this information gets conveyed to $Reader\text{-}Writer_i$.

Therefore, we may apply Lemma 8.9, with the $t$ in that theorem instantiated to $\max(t, t')$, to conclude that by time $\max(t, t') + 2d$, $cmap(k)_j \neq \perp$. This yields the conclusion. □

The following lemma specializes the previous ones to members of the reported configuration.

**Lemma 8.11** *Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$ satisfying e-recon-readiness and e-join-connectivity, $i, j \in I$, $t \in \mathbb{R}^{\geq 0}$. Suppose:*

1. *A $\mathsf{report}(c)_i$ occurs at time $t$, where $c = rec\text{-}cmap(k)_i$ and $i$ does not fail by time $t + d$.*

2. *$j \in members(c)$ and $j$ does not fail by time $t + 2d$.*

*Then by time $t + 2d$, $cmap(k)_j \neq \perp$.*

**Proof.** If $k = 0$ then the conclusion is immediate because $cmap(0)_j \neq \perp$ in all reachable states. So suppose that $k > 0$. Then a $\mathsf{join\text{-}ack}_j$ must occur by time $t - e$ by Lemma 8.7. Then Theorem 8.10, applied with $t$ and $t'$ in the statement of that theorem set to the current $t$, implies the conclusion. □

The following theorem does not use the assumption of join-connectivity. It considers the set $J$ of processes that join the system by a certain time $t$. It says that, after a time that is logarithmic in $|J|$, all the processes in $J$ know about each other, and thereafter, information about configurations propagates quickly among processes in $J$. The result assumes that the execution is failure-free (so the set of joiners cannot become partitioned).

**Theorem 8.12** *Let $\alpha$ be a normal admissible timed failure-free execution of $\mathcal{S}'$, $i, j \in I$, $J \subseteq I$, $t, t' \in \mathbb{R}^{\geq 0}$ and $t \leq t'$. Assume*

1. *$J$ is the set of processes $i'$ such that* join-ack(rambo)$_{i'}$ *occurs by time $t$.*
2. *$i, j \in J$.*

*Then*

1. *By time $t + d\lceil\log(|J|)\rceil$, $i \in world_j$.*
2. *If by time $t'$, $cmap(k)_i \neq \bot$, then by time $\max(t + d\lceil\log(|J|)\rceil, t') + 2d$ $cmap(k)_j \neq \bot$.*
3. *If by time $t'$, $cmap(k)_i = \pm$, then by time $\max(t + d\lceil\log(|J|)\rceil, t') + 2d$ $cmap(k)_j = \pm$.*

**Proof.** We show this using a pointer-doubling argument. In any state of the execution, consider the graph whose nodes are the indices of the processes that successfully joined and whose edges are the pairs $(i', j')$ such that $j' \in world_{i'}$. Since we have assumed that no failures occur, this graph is connected (this can be shown by induction on the number of joins). For the purpose of the pointer-doubling argument, process $i'$ is considered to have a "pointer" to $j'$ when $j' \in world_{i'}$.]]] Given our assumptions about the gossip, all processes that join by time $t$ require at most $\lceil\log(|J|)\rceil$ rounds of gossip to learn about all other such processes. This is because during each period of $d$ time after $t$ a round of gossip completes where at least one "pointer-doubling" occurs at each process in $J$. Information in $cmap_i$ at time $t'$ is then reflected in $cmap_j$ by time $\max(t + d\lceil\log(|J|)\rceil, t') + 2d$. □

### 8.5.3 Garbage collection

The results of this section 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 first lemma says that, assuming *$5d$-configuration-viability*, following the report of a new configuration, at least one member of the immediately preceding configuration does not fail for $4d$ time.

**Lemma 8.13** *Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$ satisfying $5d$-configuration-viability, $c \in C$, $k \in \mathbb{N}$, $k \geq 1$, $i, j \in I$, $t \in \mathbb{R}^{\geq 0}$. Suppose:*

1. *A* report$(c)_i$ *event occurs at time $t$ in $\alpha$, where $c = rec\text{-}cmap(k)_i$.*
2. *$c'$ is configuration $k-1$ in $\alpha$.*

*Then there exists $j \in members(c')$ such that $j$ does not fail by time $t + 4d$.*

**Proof.** The behavior of *Recon* algorithm implies that the time at which $Recon_i$ learns about $c$ being configuration $k$ is not more than $d$ after the time of the last $\mathsf{decide}_{k,c,*}$ event in $\alpha$. Once $Recon_i$ learns about $c$, it performs the $\mathsf{report}(c)_i$ event without any further time-passage. Then $5d$-viability ensures that at least one member of $c'$ does not fail by time $t + 4d$. $\square$

The following key lemma says that a process that has joined sufficiently long before a particular $\mathsf{report}(c)_*$ event manages to garbage collect all configurations earlier than $c$ within time $6d$ after the report.

**Lemma 8.14** *Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$ satisfying e-recon-readiness, e-join-connectivity, $6d$-recon-spacing and $5d$-configuration-viability, $c \in C$, $k \in \mathbb{N}$, $i, j \in I$, $t \in \mathbb{R}^{\geq 0}$. Suppose:*

1. *A $\mathsf{report}(c)_i$ event occurs at time $t$ in $\alpha$, where $c = \textit{rec-cmap}(k)_i$.*
2. *$\mathsf{join\text{-}ack}(\mathsf{rambo})_j$ occurs in $\alpha$ by time $t - e$.*

*Then:*

1. *If $k > 0$ and $j$ does not fail by time $t + 2d$, then by time $t + 2d$: (a) $cmap(k-1)_j \neq \perp$ and (b) $cmap(\ell)_j = \pm$ for all $\ell < k - 1$.*
2. *If $i$ does not fail by $t + d$ and $j$ does not fail by time $t + 6d$, then by time $t + 6d$: (a) $cmap(k)_j \neq \perp$ and (b) $cmap(\ell)_j = \pm$ for all $\ell < k$.*

**Proof.** By induction on $k$.
*Base:* $k = 0$.
Part 1 is vacuously true. The clause $(a)$ of Part 2 follows because $cmap(0)_j \neq \perp$ in all reachable states, and the clause $(b)$ is vacuously true.
*Inductive step:* Assume $k \geq 1$, assume the conclusions for indices $\leq k - 1$, and show them for $k$. Fix $c$, $i$, $j$, $t$ as above.
*Part 1:* Assume the hypotheses of Part 1, that is, that $k > 0$ and that $j$ does not fail by time $t + 2d$. If $k = 1$ then the conclusions are easily seen to be true: for clause $(a)$, $cmap(0)_j \neq \perp$ in all reachable states, and the clause $(b)$ of the claim is vacuously true. So from now on in the proof of Part 1, we assume that $k \geq 2$.

Since $c$ is the $k^{th}$ configuration and $k \geq 1$, the given $\mathsf{report}(c)_i$ event is preceded by a $\mathsf{recon}(*, c)_*$ event. Fix the first $\mathsf{recon}(*, c)_*$ event, and suppose it is of the form $\mathsf{recon}(c', c)_{i'}$. Then $c'$ must be the $k - 1^{st}$ configuration. Lemma 8.13 implies that at least one member of $c'$, say, $i''$, does not fail by time $t + 4d$.

The $\mathsf{recon}(c', c)_{i'}$ event must be preceded by a $\mathsf{report}(c')_{i'}$ event. Since $k - 1 \geq 1$, Lemma 8.7 implies that a $\mathsf{join\text{-}ack}(\mathsf{rambo})_{i''}$ event occurs at least time $e$ prior to the $\mathsf{report}(c')_{i'}$ event. Then by inductive hypothesis, Part 2, by time $time(\mathsf{report}(c')_{i'}) + 6d$, $cmap(k-1)_{i''} \neq \perp$ and $cmap(\ell)_{i''} = \pm$ for all $\ell < k - 1$. By *$6d$-recon-spacing*, $time(\mathsf{recon}(c', c)_{i'}) \geq time(\mathsf{report}(c')_{i'}) + 6d$, and so $t = time(\mathsf{report}(c)_i) \geq time(\mathsf{report}(c')_{i'}) + 6d$. Therefore, by time $t$, $cmap(k-1)_{i''} \neq \perp$ and $cmap(\ell)_{i''} = \pm$ for all $\ell < k - 1$.

Now we apply Lemma 8.9 to $i''$ and $j$, with $t$ in the statement of Lemma 8.9 set to the current $t$. This allows us to conclude that, by time $t + 2d$, $cmap(k-1)_j \neq \perp$ and $cmap(\ell)_j = \pm$ for all $\ell < k - 1$. This is as needed for Part 1.

*Part 2:* (Recall that we are assuming here that $k \geq 1$.) Assume the hypotheses of Part2, that is, that $i$ does not fail by time $t+d$ and $j$ does not fail by time $t+6d$. Theorem 8.10 applied to $i$ and $j$ and with $t$ and $t'$ both instantiated as the current $t$, implies that by time $t+2d$, $cmap(k)_j \neq \perp$. Part 1 implies that by time $t+2d$, $cmap(\ell)_j = \pm$ for all $\ell < k-1$. It remains to bound the time for $cmap(k-1)_j$ to become $\pm$.

By time $t+2d$, $j$ initiates a garbage-collection for $k-1$ (unless $cmap(k-1)_j$ is already $\pm$). This terminates within time $4d$. After garbage-collection, $cmap(\ell)_j = \pm$ for all $\ell < k$, as needed. The fact that this succeeds depends on quorums of configuration $k-1$ remaining alive throughout the first phase of the garbage-collection. $5d$-viability ensures this.

The calculation for $5d$ is as follows: $t$ is at most $d$ larger than the time of the last decide for configuration $k$. The time at which the garbage-collection is started is $\leq t+2d$. Thus, at most $3d$ time may elapse from the last decide for configuration $k$ until the garbage-collection operation begins. Then an additional $2d$ time suffices to complete the first phase of the garbage-collection. □

The following lemma specializes the previous one to members of the newly-reported configuration.

**Lemma 8.15** *Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$ satisfying e-recon-readiness, e-join-connectivity, $6d$-recon-spacing and $5d$-configuration-viability, $c \in C$, $k \in \mathbb{N}$, $i, j \in I$, $t \in \mathbb{R}^{\geq 0}$. Suppose:*

1. *A* report$(c)_i$ *event occurs at time $t$ in $\alpha$, where $c = rec\text{-}cmap(k)_i$.*
2. *$j \in members(c)$.*

*Then:*

1. *If $k > 0$ and $j$ does not fail by time $t+2d$, then by time $t+2d$, $cmap(k-1)_j \neq \perp$ and $cmap(\ell)_j = \pm$ for all $\ell < k-1$.*
2. *If $i$ does not fail by $t+d$ and $j$ does not fail by time $t+6d$, then by time $t+6d$, $cmap(k)_j \neq \perp$ and $cmap(\ell)_j = \pm$ for all $\ell < k$.*

**Proof.** If $k = 0$, the conclusions follow easily. If $k = 1$, then Lemma 8.7 implies that join-ack(rambo)$_j$ occurs in $\alpha$ by time $t - e$. Then the conclusions follow from Lemma 8.14. □

The following theorem says that, in the "normal case", all processes that have joined sufficiently long ago know either the latest configuration or the one just before the latest. Since we have not yet written out a proof of this, we call it a "strong conjecture".

**Theorem 8.16** *(Strong conjecture) Let $\alpha$ be a normal admissible timed execution of $\mathcal{S}'$ satisfying e-recon-readiness, e-join-connectivity, $6d$-recon-spacing and $5d$-configuration-viability, $\alpha'$ a finite prefix of $\alpha$, $k \in \mathbb{N}$, $c \in C$, $i \in I$. Suppose:*

1. *$k$ is the latest configuration index and $c$ is the latest configuration identifier, after $\alpha'$.*
2. join(rambo)$_i$ *occurs before time $\ell time(\alpha') - (e+2d)$.*
3. *$cmap(\ell)_i \in C$ just after $\alpha'$.*

*Then $\ell \in \{k-1, k\}$.*

### 8.5.4 Reads and writes

The final 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.

**Theorem 8.17** *Let $\alpha$ be a normal admissible timed execution of $S'$ satisfying e-recon-readiness, e-join-connectivity, $(12d+\varepsilon)$-recon-spacing, $11d$-configuration-viability, and infinite reconfiguration,*[8] *$i \in I$, and $t \in \mathbb{R}^+$. Assume that*

1. *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.** Let $c_0, c_1, c_2, \ldots$ denote the infinite sequence of successive configurations decided upon in $\alpha$; by infinite reconfiguration, this sequence exists. For each $k \geq 0$, let $\pi_k$ be the first $\mathsf{recon}(c_k, c_{k+1})_*$ event in $\alpha$, let $i_k$ be the location at which this occurs, and let $\phi_k$ be the corresponding, preceding $\mathsf{report}(c_k)_{i_k}$ event. (The special case of this notation for $k = 0$ is consistent with our usage elsewhere.) Also, for each $k \geq 0$, choose $s_k \in \mathit{members}(c_k)$ such that $s_k$ does not fail by time $10d$ after the time of $\phi_{k+1}$. The fact that this is possible follows from $11d$-viability (because the $\mathsf{report}$ event $\phi_{k+1}$ happens at most time $d$ after the final $\mathsf{decide}$ for configuration $k+1$).

We show that the time for each phase of the read or write operation is $\leq 4d$—this will yield the bound we need.. Consider one of the two phases, and let $\psi$ be the $\mathsf{read}_i$, $\mathsf{write}_i$ or $\mathsf{query\text{-}fix}_i$ event that begins the phase.

We claim that $\mathit{time}(\psi) > \mathit{time}(\phi_0) + 8d$, that is, that $\psi$ occurs more than $8d$ time after the $\mathsf{report}(0)_{i_0}$ event: We have that $\mathit{time}(\psi) \geq t$, and $t > \mathit{time}(\mathsf{join\text{-}ack}_i) + 8d$ by assumption. Also, $\mathit{time}(\mathsf{join\text{-}ack}_i) \geq \mathit{time}(\mathsf{join\text{-}ack}_{i_0})$. Furthermore, $\mathit{time}(\mathsf{join\text{-}ack}_{i_0}) \geq \mathit{time}(\phi_0)$, that is, when $\mathsf{join\text{-}ack}_{i_0}$ occurs, $\mathsf{report}(0)_{i_0}$ occurs with no time passage. Putting these inequalities together we see that $\mathit{time}(\psi) > \mathit{time}(\phi_0) + 8d$.

Fix $k$ to be the largest number such that $\mathit{time}(\psi) > \mathit{time}(\phi_k) + 8d$. The claim in the preceding paragraph shows that such $k$ exists.

Next, we claim that by $\mathit{time}(\phi_k) + 6d$, $\mathit{cmap}(k)_{s_k} \neq \bot$ and $\mathit{cmap}(\ell)_{s_k} = \pm$ for all $\ell < k$; this follows from Lemma 8.15, Part 2, applied with $i = i_k$ and $j = s_k$, because $i_k$ does not fail before $\pi_k$, and because $s_k$ does not fail by time $10d$ after $\phi_{k+1}$.

Next, we show that in the pre-state of $\psi$, $\mathit{cmap}(k)_i \neq \bot$ and $\mathit{cmap}(\ell)_i = \pm$ for all $\ell < k$: We apply Lemma 8.9 to $s_k$ and $i$, with $t$ in that lemma set to $\max(\mathit{time}(\phi_k) + 6d, \mathit{time}(\mathsf{join\text{-}ack}_i) + e)$. This yields that, by time $\max(\mathit{time}(\phi_k) + 6d, \mathit{time}(\mathsf{join\text{-}ack}_i) + e) + 2d$, $\mathit{cmap}(k)_i \neq \bot$ and $\mathit{cmap}(\ell)_i = \pm$ for all $\ell < k$. Our choice of $k$ implies that $\mathit{time}(\phi_k) + 8d < \mathit{time}(\psi)$. Also, by assumption, $\mathit{time}(\mathsf{join\text{-}ack}_i) + e + 2d < t$. And $t \leq \mathit{time}(\psi)$. So, $\mathit{time}(\mathsf{join\text{-}ack}_i) + e + 2d < \mathit{time}(\psi)$. Putting these inequalities together, we obtain that $\max(\mathit{time}(\phi_k) + 6d, \mathit{time}(\mathsf{join\text{-}ack}_i) + e) + 2d < \mathit{time}(\psi)$. It follows that, in the pre-state of $\psi$, $\mathit{cmap}(k)_i \neq \bot$ and $\mathit{cmap}(\ell)_i = \pm$ for all $\ell < k$, as needed.

Now, by choice of $k$, we know that $\mathit{time}(\psi) \leq \mathit{time}(\phi_{k+1}) + 8d$. The recon-spacing condition implies that $\mathit{time}(\pi_{k+1})$ (the first $\mathsf{recon}$ event that requests the creation of the $(k+2)^{nd}$ configuration) is $> \mathit{time}(\phi_{k+1}) + 12d$. Therefore, for an interval of time of length $> 4d$ after $\psi$, the largest index of any configuration that appears anywhere in the system is $k+1$. This implies that the phase of the read or write operation that starts with $\psi$ completes with at most one additional delay (of $2d$)

[8] This is assumed for simplicity, to avoid cases in the result and proof.

for learning about a new configuration. This yields a total time of at most $4d$ for the phase, as we claimed.

We use $11d$-viability here: First at most time $d$ elapses from the last $\mathsf{decide}_{k+1,*,*}$ until $\phi_{k+1}$. Then at most $8d$ time elapses from $\phi_{k+1}$ until $\psi$. At $time(\psi)$, configuration $k$ is already known (but configuration $k+1$ may not be known). Therefore we need a quorum of configuration $k$ to stay alive only for the first $2d$ time of the phase. Altogether yielding $11d$. $\square$

# 9 Latency Bounds: Normal Behavior From Some Point On

In this section, we present latency bounds for executions that exhibit normal timing and failure behavior after some point. These results correspond to some of those in Section 8, but the hypotheses and conclusions take into account the time when normal behavior begins.

## 9.1 Restricting nondeterminism

As we observed in Section 8, RAMBO is highly nondeterministic. For the purpose of the latency analysis in this section, we restrict the nondeterminism of RAMBO precisely as described in Section 8.1.

## 9.2 Normal behavior from some point on

As in Section 8, the results in this section require restrictions on timing and failure behavior—things that are not generally considered to be under the control of the algorithm. In this section, we impose timing and failure assumptions after some point in the execution, rather than throughout the execution as in Section 8.2. Each of these assumptions is, formally, a property of an admissible timed execution $\alpha$ and a finite prefix $\alpha'$ of $\alpha$. Arbitrary asynchrony is allowed in $\alpha'$, after which normal behavior holds. Specifically, we assume:

- *Normal execution after a finite prefix*: If $\alpha$ is an admissible timed execution and $\alpha'$ is a finite prefix of $\alpha$, then $\alpha$ is *$\alpha'$-normal* if the following conditions hold:

    1. *Regular timing behavior for* RAMBO *automata after $\alpha'$:* The local clocks of all $Joiner_i$, $Reader\text{-}Writer_i$, and $Recon_i$ automata progress at exactly the rate of real time, after $\alpha'$. This single assumption implies that the timing of all locally-controlled events observes real-time constraints, after $\alpha'$.
    2. *Reliable message delivery after $\alpha'$:* No message sent in $\alpha$ after $\alpha'$ is lost. (However, messages sent in $\alpha'$ may be lost.)
    3. *Message delay bound:* If a message is sent at time $t$ in $\alpha$ and it is delivered, then it is delivered by time $\max(t, \ell time(\alpha')) + d$.
    4. *Normal timing for consensus:* Timing for all consensus services is "normal" after $\alpha'$.

    These assumptions correspond to the assumptions defined in Section 8.2, which are used for analyzing the case where the entire execution $\alpha$ is normal.

As before, some of our results will also require assumptions about certain processes not failing, for certain intervals of time. Again, we state such assumptions where they are needed.

### 9.3 Hypotheses for latency results

This subsection contains one more hypothesis that we need for our latency bound results. It is needed in addition to the restrictions on nondeterminism described in Section 9.1, the behavior assumptions described in Section 9.2, and some of the properties defined for the "normal behavior" case in Section 8.3.

The new hypothesis, *join-connectivity*, is designed to ensure that all non-failing joining processes retain the ability to learn about each other. Join-connectivity is defined in terms of a *join-connectivity digraph* $JC$, which is defined as a derived variable of the system $\mathcal{S}'$:

- $JC$, the *join-connectivity digraph:* This is the digraph with self-loops defined as follows:
  1. The nodes of $JC$ are all $i \in I$ such that $\mathit{Reader\text{-}Writer}_i.\mathit{status} = \mathsf{active}$ and $\neg \mathit{Reader\text{-}Writer}_i.\mathit{failed}$.
  2. The edges of $JC$ are the pairs $(i,j) \in I \times I$ such that $j \in \mathit{Reader\text{-}Writer}_i.\mathit{world}$.

Now we define join-connectivity:

- *Join-connectivity*: We say that $\alpha$ satisfies *join-connectivity* provided that for any state $s$ occurring in $\alpha$, digraph $s.JC$ is connected[9].

### 9.4 Bounds that do not depend on gossip after stabilization

We now present performance results that do not depend on gossip after the timing and failure behavior stabilizes. More precisely, we consider the same protocol as before (see Section 8.1), in which the messages are sent when they are important and are gossiped periodically according to local clocks. However, the results of this section do not depend on gossip messages that are sent after time $\ell time(\alpha') + d$.

#### 9.4.1 Message latency

We begin with a simple lemma saying that messages that are sent in $\alpha'$ are received within a short time after the end of $\alpha'$. This follows from our assumptions about periodic gossip.

**Lemma 9.1** *Let $\alpha$ be an $\alpha'$-normal admissible timed execution of $\mathcal{S}'$, $i,j \in I$, $t \in \mathbb{R}^{\geq 0}$, and $t \leq \ell time(\alpha')$. Assume that $\mathsf{fail}_i$ and $\mathsf{fail}_j$ events do not occur in $\alpha$. Then:*

1. *If $\mathsf{send}(\mathsf{join})_{i,j}$ occurs in $\alpha'$ at time $t$, then $\mathsf{recv}(\mathsf{join})_{i,j}$ occurs in $\alpha$ by time $\ell time(\alpha') + 2d$.*
2. *If $\mathsf{send}(\langle W, *, tg, cm, pns, pnr\rangle)_{i,j}$ occurs in $\alpha'$ at time $t$, then $\mathsf{recv}(\langle W', *, tg', cm', pns', pnr'\rangle)_{i,j}$ occurs in $\alpha$ by time $\ell time(\alpha') + 2d$, where $W \subseteq W'$, $tg \leq tg'$, $cm(h) \leq cm'(h)$ for all $h \in \mathbb{N}$, $pns \leq pns'$, and $pnr \leq pnr'$.*
3. *If $\mathsf{send}(\langle \mathsf{config}, c, k\rangle)_{i,j}$ occurs in $\alpha'$ at time $t$, then $\mathsf{recv}(\langle \mathsf{config}, c, k\rangle)_{i,j}$ occurs in $\alpha$ by time $\ell time(\alpha') + 2d$.*
4. *If $\mathsf{send}(\langle \mathsf{init}, c, c', k\rangle)_{i,j}$ occurs in $\alpha'$ at time $t$, then $\mathsf{recv}(\langle \mathsf{init}, c, c', k\rangle)_{j,i}$ occurs in $\alpha$ by time $\ell time(\alpha') + 2d$.*

[9]That is, the undirected version of the join-connectivity digraph, in which every directed edge is converted to an undirected edge, is connected.

**Proof.** Parts 1, 3, and 4 follow directly from the gossip policy and the assumption that $\alpha$ is $\alpha'$-*normal* and admissible. Since $i$ does not fail, it gossips all messages of the types $\mathsf{join}$, $\mathsf{init}$, and $\mathsf{config}$. At least one instance of gossip for each message type must occur after $\alpha'$ and by time $\ell time(\alpha') + d$. Since $j$ does not fail, it receives at least one such message by time $\ell time(\alpha') + 2d$.

Part 2 is similar, except that the required relations between the message components must hold. This is shown by observing that all changes to the relevant state components are monotone. If the original message is not lost, then the received message may be taken to be the same as the one that is sent at time $t$, which implies that the components of the received message are equal to those of the one originally sent. On the other hand, if the original message is lost, then a subsequently gossiped message is received by the indicated time, and its components are not smaller than those in the original message. $\square$

### 9.4.2 Joining

The next theorem implies that if the creator starts the join protocol with the $\mathsf{join}(\mathsf{rambo}, J)_{i_0}$ event at time $t$, then it finishes by time $\max(t, \ell time(\alpha'))$, provided $i_0$ does not fail. Also, if a non-creator starts the join protocol at time $t$, it finishes by time $\max(t, \ell time(\alpha')) + 3d$, provided the relevant processes do not fail.

**Theorem 9.2** *Let $\alpha$ be an $\alpha'$-normal admissible timed execution of $\mathcal{S}'$. If $\mathsf{join}(\mathsf{rambo}, J)_i$ occurs in $\alpha$ at time $t$ and $\mathsf{fail}_i$ does not occur then:*

1. *If $i = i_0$ then $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ occurs by time $\max(t, \ell time(\alpha'))$.*
2. *Suppose that $i \neq i_0$. Suppose also that, for some $j \in J - \{i\}$, a $\mathsf{join\text{-}ack}(\mathsf{rambo})_j$ event occurs prior to the $\mathsf{join}(\mathsf{rambo}, J)_i$ event, and $\mathsf{fail}_j$ does not occur. Then:*
   - *(a) If $\mathsf{join}(\mathsf{rambo}, J)_i$ occurs in $\alpha'$ then $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ occurs by time $\ell time(\alpha') + 3d$.*
   - *(b) If $\mathsf{join}(\mathsf{rambo}, J)_i$ occurs after $\alpha'$ then $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ occurs by time $t + 2d$.*

**Proof.** Similar to the proof of Theorem 8.1. Part 1 immediately follows from the code for the creator: $Joiner_{i_0)}$, $Reader\text{-}Writer_{i_0}$ and $Recon_{i_0}$. This is because the response does not depend on the receipt of any messages.

We now consider Part 2(a). If the $\mathsf{join}_i$ event occurs in $\alpha'$ then it is possible that process $i$'s initial $\mathsf{join}$ message to $j$ is lost; however, within time $d$ of the end of $\alpha'$, $i$ is guaranteed to resend the message, and this new message is guaranteed to be received by $j$ by time $\ell time(\alpha') + 2d$ (by Lemma 9.1). Since $j \in J$ and $j$ does not fail, and since $\mathsf{join\text{-}ack}(\mathsf{rambo})_j$ occurs prior to $\mathsf{join}(\mathsf{rambo}, J)_i$, it follows that $j$ must respond to such a join message by $\ell time(\alpha') + 2d$, and this response is received by $i$ by $\ell time(\alpha') + 3d$.

Finally, we consider Part 2(b). If the $\mathsf{join}_i$ event occurs after $\alpha'$, then at most two message delays are incurred by the protocol, since no messages are lost and since $j$ is guaranteed to respond. Thus $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ occurs by time $t + 2d$. $\square$

### 9.4.3 Reconfiguration

We show that if process $i$ starts a reconfiguration with a $\mathsf{recon}(c,*)_i$ event at time $t$ and no failure event occur among the members of $c$, then reconfiguration completes by time $\max(t, \ell time(\alpha')) + 12d + \varepsilon$. (We let $\varepsilon$ be fixed as needed for Theorem 7.2.)

In the setting with arbitrary initial behavior, we cannot characterize interesting executions in terms of *e-configuration-viability* for a fixed $e$ because an arbitrary amount of time may elapse from when a configuration becomes installed until it is garbage-collected. Therefore, in the rest of Section 9, we limit our consideration to executions in which no quorum system is ever disabled, that is, executions satisfying $\infty$-*configuration-viability*.

**Theorem 9.3** *Let $\alpha$ be an $\alpha'$-normal admissible timed execution of $\mathcal{S}'$ satisfying $\infty$-configuration-viability, and let $t \in \mathbb{R}^{\geq 0}$. Assume that:*

1. *A $\mathsf{recon}(c,c')_i$ event occurs at time $t$ in $\alpha$.*
2. *No* fail *event for a member of $c$ occurs in $\alpha$ after the $\mathsf{recon}(c,c')_i$ event.*

*Then a $\mathsf{recon\text{-}ack}(*)_i$ event matching the assumed $\mathsf{recon}(c,c')_i$ event occurs by time $\max(t, \ell time(\alpha')) + 12d + \varepsilon$.*

**Proof.** The proof follows the pattern established in Theorem 8.2. Let $t_1 = \max(t, \ell time(\alpha'))$.

We know that $i$, the originator of the operation, does not fail, because the signature restrictions for *Recon* require that $i \in members(c)$, and assumption 2 says that no members of $c$ fail after the $\mathsf{recon}(c,c')_i$ event.

When $\mathsf{recon}(c,c')_i$ occurs, if *outcome* is immediately set to nok, then the $\mathsf{recon\text{-}ack}(\mathsf{nok})_i$ event occurs by time $t_1$. On the other hand, if *outcome* is not immediately set to nok, then process $i$ sets *cons-data*$_i$ in preparation for consensus, again by time $t_1$. Then an $\mathsf{init}(c')_{k,c,i}$ event occurs for some $k$, again by time $t_1$. All these events must occur by time $t_1$ because $\mathsf{recon}(c,c')_i$ occurs by time $t_1$ and $t_1 \geq \ell time(\alpha')$.

Then we claim that a $\mathsf{decide}(*)_{k,c,i}$ event occurs by time $t_1 + 12d + \varepsilon$, and subsequently $\mathsf{recon\text{-}ack}(*)_i$ occurs, also by time $t_1 + 12d + \varepsilon$. The argument that $\mathsf{decide}(*)_{k,c,i}$ occurs by time $t_1 + 12d + \varepsilon$ proceeds as follows:

First, the last init event for $Cons(k,c)$ that occurs in $\alpha$ must occur by time $t_2 = t_1 + 2d$. This is guaranteed by the sending of init messages followed by the gossip of these messages within *Recon* (by Lemma 9.1). Let $\alpha''$ be the shortest prefix of $\alpha$ that extends $\alpha'$ and includes all the $\mathsf{init}(*)_{k,c,*}$ events that occur in $\alpha$. Then we know that $\ell time(\alpha'') \leq t_2$.

We will apply Theorem 7.2 to $\alpha$ and $\alpha''$ (using $\alpha''$ for the $\alpha'$ of that theorem) to conclude that by time $t_2 + 10d + \varepsilon = t_1 + 12d + \varepsilon = \max(t, \ell time(\alpha')) + 12d + \varepsilon$, a $\mathsf{decide}(*)_{k,c,j}$ event occurs for every non-failed $j \in members(c)$. In particular, a $\mathsf{decide}(*)_{k,c,i}$ event occurs by time $\max(t, \ell time(\alpha')) + 12d + \varepsilon$. If the $\mathsf{decide}(*)_{k,c,i}$ event occurs in $\alpha'$, then the corresponding recon-ack event occurs by $\ell time(\alpha')$, which suffices. On the other hand, if the $\mathsf{decide}(*)_{k,c,i}$ event occurs after $\alpha'$, then the recon-ack event happens within time 0 of the decide event, which again suffices.

It remains to show that the three hypotheses of Theorem 7.2 are satisfied. For Property 1, the "normal case" assumptions of this section imply that timing is regular and no message losses occur after $\alpha''$. No process failures occur either: since the $\mathsf{init}(c')_{k,c,i}$ event follows the $\mathsf{recon}(c,c')_i$ event, an assumption of this theorem implies that no fail events for members of $c$ occur in $\alpha$ after $\alpha''$.

The argument for Property 2 is exactly the same as in Theorem 8.2. Property 3 says that some read-quorum and some write-quorum of $c$ must stay non-failed forever. This is guaranteed by the $\infty$-*configuration-viability* assumption. $\square$

### 9.4.4 Garbage collection

We show that for $\infty$-*configuration-viable* executions of $\mathcal{S}'$, if a garbage-collection operation starts at time $t$, it finishes by time $\max(t, \ell time(\alpha')) + 5d$. In the theorem statement, we explicitly assume the existence of certain non-failing quorums rather than assuming $\infty$-*configuration-viability*.

**Theorem 9.4** *Let $\gamma$ be a garbage-collection operation in an $\alpha'$-normal admissible timed execution of $\mathcal{S}'$. Let $\gamma$ start with $\mathsf{gc}(k)_i$ at time $t$ and let $c_k$ and $c_{k+1}$ be the values of $cmap(k)_i$ and $cmap(k+1)_i$ when $\gamma$ starts.*
*Let $R \in read\text{-}quorums(c_k)$, $W_1 \in write\text{-}quorums(c_k)$, $W_2 \in write\text{-}quorums(c_{k+1})$. Assume:*

1. *Process $i$ does not fail.*

2. *No process in $R \cup W_1 \cup W_2$ fails.*

*Then $\gamma$ ends with a $\mathsf{gc\text{-}ack}(k)_i$, by time $\ell time(\alpha') + 5d$ if the $\mathsf{gc}(k)_i$ event occurs in $\alpha'$, and by time $t + 4d$ if the $\mathsf{gc}(k)_i$ event occurs after $\alpha'$.*

**Proof.** The case where the $\mathsf{gc}(k)_i$ event occurs after $\alpha'$ is the same as Theorem 8.4. We consider the case where $\mathsf{gc}(k)_i$ event occurs in $\alpha'$ in detail:

Garbage collection is implemented in two phases. In the first phase, process $i$ sends messages to $members(c(k))$ and collects responses. If messages from $i$ are sent in $\alpha'$ they may be lost. However, such messages are subsequently gossiped. At least one round of gossip with no message loss occurs by time $\max(t, \ell time(\alpha')) + d$. These messages are delivered by time $\max(t, \ell time(\alpha')) + 2d$, by Lemma 9.1. With assumption 2, this ensures that $i$ receives the necessary responses by time $t_1 = \max(t, \ell time(\alpha')) + 3d$. Lemma 9.1 also insures that the phase number component of the replies is at least as high as the phase of the garbage-collection operation.

The second phase is similar, except that $i$ communicates with $members(c(k+1))$. If the second phase starts in $\alpha'$ then it is guaranteed to complete by time $\ell time(\alpha') + 3d$ (again using Lemma 9.1). If the second phase starts after $\alpha'$, then it must start without delay after the end of the first phase, and no later than the time $t_1$. This means that in this case the second phase completes by time $t_1 + 2d$. This is due to the two message delays incurred in this phase. Then assumption 2 ensures that $i$ receives the necessary responses. Combining these time bounds gives the result that garbage collection completes by time $\max(t, \ell time(\alpha')) + 5d$. $\square$

Note that if an execution satisfies $\infty$-*configuration-viability*, then assumption 2 of the theorem holds for any configuration. In this case, if a garbage collection starts at time $t$ and the initiator does not fail, the garbage-collection completes successfully by time $\max(t, \ell time(\alpha')) + 5d$.

## 9.5 Bounds that depend on gossip throughout execution

We now show performance results that depends on periodic gossip throughout the entire execution of system $\mathcal{S}'$.

### 9.5.1 Learning about participants and configurations

The following theorem uses the assumption of *join-connectivity*. It considers the set $J$ of processes that join the system by a certain time $t$. It says that, after a time that is logarithmic in $|J|$ following $\alpha'$, all the processes in $J$ know about each other, and thereafter, information about configurations propagates quickly among processes in $J$. The result assumes that processes in $J$ do not fail after time $t$, to ensure rapid propagation of information.

**Theorem 9.5** *Let $\alpha$ be an $\alpha'$-normal admissible timed execution of $S'$ satisfying join-connectivity, and let $J \subseteq I$, $i, j \in J$, $t, t' \in \mathbb{R}^{\geq 0}$ and $t \leq t'$. Assume*

1. *$J$ is the set of processes $i'$ such that* $\mathsf{join\text{-}ack(rambo)}_{i'}$ *occurs by time $t$.*
2. *No* $\mathsf{fail}_{i'}$ *events for $i' \in J$ occur in $\alpha$.*

*Then*

1. *By time* $\max(t, \ell time(\alpha')) + d + d\lceil\log(|J|)\rceil$*, $i \in world_j$.*
2. *If by time $t'$, $cmap(k)_i \neq \bot$, then by time* $\max\left(\max(t, \ell time(\alpha')) + d + d\lceil\log(|J|)\rceil, t'\right) + 2d$ *$cmap(k)_j \neq \bot$.*
3. *If by time $t'$, $cmap(k)_i = \pm$, then by time* $\max\left(\max(t, \ell time(\alpha')) + d + d\lceil\log(|J|)\rceil, t'\right) + 2d$ *$cmap(k)_j = \pm$.*

**Proof.** The proof follows that of Theorem 8.12, but with the failure-free assumption replaced with a weaker *join-connectivity* assumption during $\alpha'$ followed by the absence of failures of processes in $J$ after $\alpha'$.

We show this using a pointer-doubling argument. For Part 1, a process $j$ is considered to have a "pointer" to $i$ when $i \in world_j$. Given our assumptions about the gossip, during each period of $d$ time after $t$ a "round" of gossip completes where at least one "pointer-doubling" occurs at each process in $J$. However messages can be lost in $\alpha'$ or have unbounded delay. Therefore periodic message-lossless pointer-doubling starts at the latest by time $\max(t, \ell time(\alpha'))+d$. The first reliable round of gossip completes by time $\max(t, \ell time(\alpha')) + 2d$ (Lemma 9.1), and thereafter will occur at least once every $d$ time. Therefore all processes that join by time $t$ require at most $\lceil\log(|J|)\rceil$ rounds of gossip to learn about all other such processes.

For Parts 2 and 3, given $t \leq t'$ and using Part 1, the information in $cmap_i$ at time $t'$ is reflected in $cmap_j$ by time $\max(\max(t, \ell time(\alpha')) + d + d\lceil\log(|J|)\rceil, t') + 2d$. Here the quantity $2d$ corresponds to a delay of $d$ from the time when $cmap_j$ changes until the next gossip round begins, and an additional delay of $d$ for the delivery of the gossip message. $\square$

### 9.5.2 Garbage collection progress

We show that, after the system stabilizes, the time needed to garbage-collect all but one configuration found in any process' truncated *cmap* is at most linear in the length of that truncated *cmap*. We state this result for a collection of processes that join by a certain time, and is shown to hold after a sufficient delay necessary for them to discover one another.

**Theorem 9.6** *Let $\alpha$ be an $\alpha'$-normal admissible timed execution of $\mathcal{S}'$ satisfying join-connectivity and $\infty$-configuration-viability, $i \in I$, $J \subseteq I$, $t, t' \in \mathbb{R}^{\geq 0}$, and $t' > \max(t, \ell time(\alpha')) + d + d\lceil \log(|J|) \rceil$. Assume*

1. *$J$ is the set of processes $i'$ such that* $\mathsf{join\text{-}ack(rambo)}_{i'}$ *occurs by time $t$.*
2. *$k = \max\{h : truncate(cmap_i)(h) \in C\}$ at time $t'$.*
3. *No* $\mathsf{fail}_{i'}$ *events for $i' \in J \cup \{i\}$ occur in $\alpha$.*

*Then $cmap(h)_i = \pm$ for all $h$ such that $0 \leq h < k$ by time $t' + 4\,d\,k$.*

**Proof.** Given assumptions about $J$ and *join-connectivity*, Theorem 9.5 establishes that processes in $J$ know about each other by time $t'$. We consider two cases.

First, we consider executions where no garbage-collection operations start before time $t'$. If $k = 0$, then garbage-collection is not enabled and the result holds at time $t'$ as required. If $k > 0$, then given that $t' > \ell time(\alpha')$ and $\infty$-*configuration-viability*, Theorem 9.4 says that each garbage-collection operation at $i$ takes $4d$ time. By the definition of $k$, we require that at most $k$ configurations are to be garbage-collected at $i$. Since garbage-collection is enabled at $i$ at time $t'$, it starts without any delay. This yields the result.

In the second case, we consider executions where 0 or more garbage-collection operations at $i$ may have completed in $\alpha'$. If no garbage-collection operations are in progress at time $t'$, then the result is obtained as in the first case. Else, if a garbage-collection operation is in progress at time $t'$ and it started after $\alpha'$, then by Theorem 9.4 it completes by time $t' + 4d$, then the result easily follows. Finally, the most interesting situation is when a garbage-collection operation is in progress at time $t'$ and it started during $\alpha'$. By Theorem 9.4 this garbage-collection operation completes by time $\ell time(\alpha') + 5d$. Then, using Theorem 9.4 again, the garbage collection of the remaining $k-1$ configurations are completed by time $\ell time(\alpha') + 5d + 4d(k-1)$. Since $t' > \ell time(\alpha') + d$ (from the theorem assumption about $t'$ and given that $\lceil \log(|J|) \rceil$ is positive for any $J$), the result follows. □

### 9.5.3 Read-write operation latency

Our final theorem describes a situation in which a read or write operation is guaranteed to have latency at most $4d$: when the configuration map of the operation's initiator contains multiple configurations, including the latest one and no new configurations are being determined. Since the configurations are used concurrently by the read or write operation, they do not slow the operation down. Here, we do not require garbage-collection, but we need to assume $\infty$-*configuration-viability*.

**Theorem 9.7** *Let $\alpha$ be an $\alpha'$-normal admissible timed execution of $\mathcal{S}'$ satisfying $\infty$-configuration-viability, $i \in I$, $J \subseteq I$, $t, t' \in \mathbb{R}^{\geq 0}$ and $t' > \max(t, \ell time(\alpha')) + d\lceil \log(|J|) \rceil + 3d$. Assume*

1. *$J$ is the set of processes $i'$ such that* $\mathsf{join\text{-}ack(rambo)}_{i'}$ *occurs by time $t$.*
2. *$\alpha$ contains no* $\mathsf{decide}$ *events after time $t$.*
3. *$k$ is the latest configuration index in $\alpha$.*
4. *No* $\mathsf{fail}_*$ *events occur at or after time $t$.*

*Then if a read or write operation starts at time $t'$ in a state where $cmap(\ell)_i \neq \perp$ for all $\ell$, $0 \leq \ell \leq k$, then it completes by time $t' + 4d$.*

**Proof.** Given assumptions about $\alpha$, $J$ and *join-connectivity*, Theorem 9.5 establishes that by time $t'$ the processes in $J$ know about each other, and about all configurations decided by time $t$. The result then follows from the two-phased implementation of operations. Each phase lasts for at most two message delays: since new configurations are not added to $op.cmap_i$ during the phase, the phase completes in $2d$ time. New configurations can only be added in the effects of the recv action in *Reader-Writer*$_i$. Because $k$ is the latest configuration index, no higher numbered configurations exist, and smaller numbered configurations cannot be added because of the properties of the *extend* and *truncate* functions used to modify $op.cmap_i$ in the effects of recv. □

## 10 Conclusions

We have presented a specification for RAMBO, a new reconfigurable atomic memory service for read/write objects, and have presented and analyzed a new, highly concurrent asynchronous message-passing algorithm that implements RAMBO. The algorithm uses a loosely-coupled reconfiguration service, which in turn uses a sequence of consensus services, one for each new configuration. Each consensus service is implemented using Paxos. The entire algorithm satisfies its safety properties in the presence of any pattern of asynchrony and failures. The performance of the algorithm depends on assumptions about message delay and failures. The limitations say, essentially, that each non-superseded configuration is "viable" (some read-quorum and some write-quorum continue to operate) for a certain amount of time.

In future work, we plan to analyze the algorithm under more sets of assumptions. Most of our analysis so far has dealt with the case where behavior is normal throughout the execution. Our results for the situation where behavior is normal from some point onward are still incomplete. In particular, we would like results that bound the latency of read and write operations that begin sufficiently long after the system has stabilized. In this paper we gave a simple bound for read and write operations in executions that include process failures, but where no configuration ever becomes disabled. We intend to show additional bounds for executions where configurations may become disabled, provided that they remain alive long enough to be garbage-collected.

The RAMBO algorithm is very nondeterministic and so it can be tuned for performance in a variety of ways, for example, by varying the frequency of gossiping and directing the gossip message to certain subsets of the participants. We plan to evaluate the impact that various choices have on the algorithm's fault-tolerance, latency, and communication costs. For example, what are the tradeoffs between the frequency of gossiping and the latency of operations? We intend to examine restrictions on gossip where a process follows a gossip policy based on whether it is a member of certain configurations. In particular, a distinct gossip policy can be prescribed for a process that is not a member of any configuration.

Also, we would like to analyze tradeoffs between the amount of time that a configuration is assumed to remain viable (that is, the quantity $e$ in the *e-viable* hypothesis) and other factors, such as the message delay $d$, the amount of process failure, and the frequency of gossip. With such tradeoffs, the knowledge about the length of time that a configuration is expected to remain alive will determine the necessary frequency of gossip, and this will lead to better performance.

In other future work, we plan to implement and test the complete RAMBO algorithm in LAN, WAN, and mobile settings, and to use these implementations to build toy applications. So far, two LAN implementations have been begun, by Peter Musial, Jon Luke, Bence Magyar, and Matt Bachmann. We will compare our theoretical results on performance analysis to experimental results obtained from these implementations.

We are also considering various improvements to the algorithm. For example, we are investigating ways of increasing the concurrency of garbage-collection. We will consider variants of the algorithm that allow early return of the results of read operations, before the propagation phase is executed. Such results are guaranteed to be the same as the results that would be returned after the propagation phase, so there appears to be little practical reason not to return them early; however, the effects of doing this need to be carefully understood. More generally, we will consider augmenting the algorithm with the capability to return "best available" versions to clients that prefer not to wait for an atomic version.

We will consider "backup" strategies for coping with the situation where viability fails, and the object therefore becomes inaccessible. For example, the system might automatically create a new "continuation" of an object for which too many configuration members fail. It might do this, for instance, by reading several copies of the object, and using the value with the largest available tag to start the new object. Questions remain about who is authorized to create such a continuation.

This work leaves open the very important question of how to choose good configurations, for various kinds of platforms.

One can also study the "join problem". As we already noted, join-connectivity is not really appropriate as a basic assumption. It remains to formulate appropriate basic assumptions and possibly improve the joining protocol, to prove a version of join-connectivity from more basic assumptions. It is possible that the join problem itself could be studied as a problem of independent interest.

**Acknowledgments.** The authors thank Ken Birman, Alan Demers, Rui Fan, Seth Gilbert, Butler Lampson and Peter Musial for helpful discussions.

## References

[1] *Communications of the ACM*, special section on group communications, vol. 39, no. 4, 1996.

[2] D. Agrawal and A. El Abbadi, "Resilient Logical Structures for Efficient Management of Replicated Data", *TR, Univ. of California Santa Barbara*, 1992.

[3] L. Alvisi, D. Malkhi, L. Pierce, and M. Reiter, "Fault detection for Byzantine quorum systems",(extended abstract), *Proc. of the 7th IFIP International Working Conference on Dependable Computing for Critical Applications*, 1999.

[4] Amir Y., Dolev P., Melliar-Smith P., Agarwal D., and Ciarfella P. "Fast Message Ordering and Membership using a Logical Token-Passing Ring". In *13th International Conference on Distributed Computing Systems (ICDCS)*, pages 551–560, 1993.

[5] Y. Amir, D. Dolev, P. Melliar-Smith and L. Moser, "Robust and Efficient Replication Using Group Communication" Technical Report 94-20, Department of Computer Science, Hebrew University., 1994.

[6] Y. Amir, A. Wool, "Evaluating Quorum Systems over the Internet", *Proc. of 26th Intl. Symp. on Fault-Tolerant Computing*, Sendai, Japan, pp. 26-35, 1996.

[7] 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.

[8] M. Bearden, R. P. Bianchini Jr., "A Fault-tolerant Algorithm for Decentralized On-line Quorum Adaptation", in *Proc. 28th Intl. Symp. on Fault-Tolerant Computing Systems*, Munich, Germany, 1998.

[9] P.A. Bernstein, V. Hadzilacos and N. Goodman, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, Reading, MA, 1987.

[10] F. Cristian and F. Schmuck, "Agreeing on Processor Group Membership in Asynchronous Distributed Systems", Technical Report CSE95-428, Dept. of Computer Science, University of California San Diego.

[11] S.B. Davidson, H. Garcia-Molina and D. Skeen, "Consistency in Partitioned Networks", *ACM Computing Surveys*, vol. 15, no. 3, pp. 341-370, 1985.

[12] A. Demers, D. Greene, A. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In Proc. ACM Symp. on the Principles of Distr. Computing, pages 1–12, August 1987.

[13] R. De Prisco, A. Fekete, N. Lynch, A. Shvartsman, "A Dynamic Primary Configuration Group Communication Service", *13th International Conference of Distributed Computing*, 1999.

[14] Roberto De Prisco, Nancy Lynch, Alex Shvartsman, Nicole Immorlica and Toh Ne Win "A Formal Treatment of Lamport's Paxos Algorithm", manuscript, 2002.

[15] C. Dwork, N. A. Lynch, L. J. Stockmeyer, "Consensus in the presence of partial synchrony", *J. of ACM*, 35(2), pp. 288-323, 1988.

[16] A. El Abbadi, D. Skeen and F. Cristian, "An Efficient Fault-Tolerant Protocol for Replicated Data Management", in *Proc. of the Fourth ACM Symp. on Princ. of Databases*, pp. 215-228, 1985.

[17] A. El Abbadi and S. Toueg, "Maintaining Availability in Partitioned Replicated Databases", *ACM Trans. on Database Systems*, vol. 14, no. 2, pp. 264-290, 1989.

[18] 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.

[19] A. Fekete, N. Lynch and A. Shvartsman "Specifying and using a partitionable group communication service", *ACM Transaction on Computer Systems*, vol. 19, no. 2, pp. 171–216, 2001.

[20] 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.

[21] D.K. Gifford, "Weighted Voting for Replicated Data", in *Proc. of 7th ACM Symp. on Oper. Sys. Princ.*, pp. 150-162, 1979.

[22] K. Goldman and N. Lynch, "Nested Transactions and Quorum Consensus", in *Proc. of the 6th ACM Symp. on Princ. of Distr. Comput.*, pp. 27-41, 1987

[23] M.P. Herlihy, "Replication Methods for Abstract Data Types", *Doctoral Dissert., MIT, LCS/TR-319*, 1984.

[24] M.P. Herlihy, "Dynamic Quorum Adjustment for Partitioned Data", *ACM Trans. on Database Systems*, 12(2), pp. 170-194, 1987.

[25] S. Jajodia and D. Mutchler, "Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database", in *ACM Trans. Database Systems*, 15(2), pp. 230-280, 1990.

[26] David Kempe, Jon M. Kleinberg, Alan J. Demers: Spatial gossip and resource location protocols. STOC 2001: 163-172.

[27] 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.

[28] I. Keidar, *A Highly Available Paradigm for Consistent Object Replication*, M.Sc. Thesis, Hebrew Univ., Jerusalem, 1994; (see also TR CS95-5 at URL: http://www.cs.huji.ac.il/~transis/publications.html).

[29] 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.

[30] Leslie Lamport, "The Part-Time Parliament", *ACM Transactions on Computer Systems*, 16(2) 133-169, 1998.

[31] M. Liu, D. Agrawal and A. El Abaddi, "On the Implementation of the Quorum Consensus protocol", *Proc. Parallel and Distributed Computing Systems*, Orlando, Florida, 1995.

[32] E. Lotem, I. Keidar, and D. Dolev, "Dynamic Voting for Consistent Primary Components", in *Proc. 16 ACM Symp. on Principles of Distributed Computing*, pp. 63-71, 1997.

[33] N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[34] Nancy Lynch and Alex Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 272–281, Seattle, Washington, USA, June 1997. IEEE.

[35] D. Malki and M. Reiter, "Byzantine Quorum Systems", in *Proceedings of the 29th ACM Symposium on Theory of Computing*, pp. 569-578, 1997.

[36] D. Peleg and A. Wool, "The Availability of Quorum Systems", *Information and Computation*, 123(2), pp. 210-223, 1995.

[37] S. Rangarajan, S. Tripathi, "A Robust Distributed Mutual Exclusion Algorithm", *Distributed algorithms, Proceedings 5th Intl. Workshop,* WDAG '91, Delphi, pp. 295-308, 1991.

[38] B. Sanders, "The Information Structure of Distributed Mutual Exclusion Algorithms", *ACM Transactions on Computer Systems*, 5(3), Aug. 1987, pp.284-299.

[39] E. Upfal and A. Wigderson, How to share memory in a distributed system, *Journal of the ACM*, 34(1):116–127, 1987.