RAMBO: A Robust, Reconfigurable Atomic Memory Service for Dynamic Networks [*]

Seth Gilbert[†] Nancy A. Lynch[‡] Alexander A. Shvartsman[§]

January 9, 2008

Abstract

This paper presents RAMBO, an algorithm that emulates read/write shared objects in a dynamic setting. To ensure that the data is highly available and long-lived, each object is replicated at several physical locations. To ensure atomicity, reads and writes are performed using *quorum configrations*, 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 a 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 completed within $8d$ time, where d is the maximum message latency. Similar results are achieved when these conditions hold *eventually* at some point in the execution, rather than throughout the entire execution. That is, the RAMBO protocol rapidly stabilizes, resuming efficient operation, during intervals in which good conditions hold.

[*]This work was supported in part by the NSF ITR Grant CCR-0121277. Preliminary versions of this work appeared as the following extended abstracts: (a) Nancy A. Lynch, Alexander A. Shvartsman: RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. DISC 2002: 173-190, and (b) Seth Gilbert, Nancy A. Lynch, Alexander A. Shvartsman: RAMBO II: Rapidly Reconfigurable Atomic Memory for Dynamic Networks. DSN 2003: 259-268.

[†]School of Computer and Communication Sciences, Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland. Email: `seth.gilbert@epfl.ch`.

[‡]Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA. Email: `lynch@theory.lcs.mit.edu`. The work of this author was additionally supported by the NSF Grant 9804665.

[§]Department of Computer Science and Engineering, 371 Fairfield Way, Unit 2155, University of Connecticut, Storrs, CT 06269, USA and Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA. Email: `aas@cse.uconn.edu`. The work of the third author was additionally supported in part by the NSF Grants 9984778, 9988304, and 0311368.

1 Introduction

This paper presents RAMBO, an algorithm that implements atomic read/write shared memory in a dynamic setting in which participants may join, leave, or fail during the course of computation. Examples of such settings include mobile networks and peer-to-peer networks where data survivability is of high concern. One use of this service might be to provide long-lived data in a dynamic and volatile setting such as a military operation or a disaster response effort.

In order to achieve availability in the presence of failures, the data 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[1]." 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) choose 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 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 the old configurations to be removed, while the second phase communicates with a configuration that remains extant. A garbage-collection operation ensures that both a read-quorum and a write-quorum of each old configuration learn about the newer configuration, and that the latest value from the old configurations is conveyed to a write-quorum of the newer configuration.

The reconfiguration service $Recon$ is implemented by a distributed algorithm that uses distributed consensus to agree on the successive configurations. Any member of the latest configuration c may propose a new configuration at any time; different proposals are reconciled by an execution of consensus among the members of c. Consensus is, in turn, implemented using a version of the Paxos algorithm [36], as described formally in [**?**]. 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 not requested too frequently, and that quorums of active configuration do not fail, we show that read and write operations complete within $8d$, where d is the max-

[1] Here "Basic" means "Read/Write", but RAMRWO would not be nearly as compelling an acronym.

imum latency. Similar results are achieved when these conditions hold *eventually* at some point in the execution, rather than throughout the entire execution. That is, the RAMBO protocol rapidly stabilizes, resuming efficient operation, during intervals in which good conditions hold.

The RAMBO protocol, in many ways, is an architectural template that has formed the basis for a variety of other algorithms. In Section 12 we summarize the results of other research on optimizations, refinements, and implementations based on RAMBO.

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 [36]. 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: read and writes are guaranteed to complete, provided that currently active configurations are not disabled by failures.

Group communication services (GCS) [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 [21] using techniques of Amir, Dolev, Keidar, Melliar-Smith and Moser [33, 34, 6]. 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 [13] 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 [16] 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 [8] 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 [16] delays reads and writes during reconfiguration.

In earlier work on atomic memory for dynamic networks [40, 20], 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 [40, 20], 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 [40, 20], 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 showed the first general scheme for emulating shared-memory in message-passing systems by using replication and accessing majorities of time-stamped replicas [48]. Attiya, Bar-Noy and Dolev developed a majority-based emulation of atomic registers that uses bounded time-stamps [8].

Their algorithm introduced a two-phase paradigm where in the first phase information is gathered from a majority of processors, and the second phase propagates information to a majority of processors.

Quorums [25] are generalizations of majorities. A *quorum system* (also called a *coterie*) is a collection of sets such that any two sets, called *quorums*, intersect [22]. Another approach is to separate quorums into 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 distributed mutual exclusion [22] and data replication protocols [14, 29]. Quorums can be used with replicated data in transaction-style synchronization that limits concurrency (cf. [11]). Many other replication techniques use quorums [2, 9, 18, 19, 26]. An additional level of fault-tolerance in quorum-based approaches can be achieved using the Byzantine quorum approach [41, 4].

A great variety of research has been carried out on the fault-tolerance of quorum assignments. Probabilistic approaches such as [7, 37, 44, 45], develop methods to determine the likelihood that progress is achieved given a non-adaptive quorum system. When processors fail with a known probability, a quorum assignment can be selected to maximize the probability of progress. This method can also be used with our emulation to allow a system monitor to evaluate the current configuration and to make decisions concerning its replacement.

Another approach to quorum adaptation is dynamic voting [30, 31, 38]. The approach in [31] relies on locking and requires that at least a majority of all the processors in some previously updated quorum (or half of all the processors in some previously updated quorum plus the distinguished site) are still alive. The approach in [38] does not rely on locking, but requires at least a predefined number of processors to always be alive. The on-line quorum adaptation of [9] assumes the use of Sanders [47] mutual exclusion algorithm, which again relies on locking.

Document structure. In Section 2 we present the models and data types used in the sequel. The specification of the RAMBO is given in Section 3. In Section 4 we overview the structure of the main technical development. In Section 5, we present the specification for the $Recon$ service. In Section 6, we present the main algorithm implementing read and write operations. In Section 7, we show that the main algorithm satisfies the safety properties described in Section 3. In Section 8, we present our implementation of $Recon$ and show that it satisfies the specified properties. Before presenting the latency analysis, we describe our timed model of computation and assumptions in Section 9. Section 10 presents results on latency and fault-tolerance under the assumption that the system is well-behaved. In Section 11, we present similar results for the case where the system may be initially badly behaved, but is eventually well-behaved. In Section 12 we summarize the results of other research on optimizations, refinements, and implementations based on RAMBO as an architectural template. We conclude in Section .

2 Model and Data Types

We use the asynchronous message-passing model, in which uniquely identified asynchronous processes communicate using point-to-point asynchronous channels. All processes may communicate with each other. Processes may fail by crashing (stopping without warning). Our safety results do not depend on any assumptions about message delivery time. However, for our performance results, we assume that messages are delivered in bounded time.

2.1 Data types

We now describe the data types used in our exposition. We assume two distinguished elements, $\perp$ and $\pm$, which are not in any of the basic types. For any type A, we define new types $A_\perp = A \cup \{\perp\}$, and $A_\pm = A \cup \{\perp, \pm\}$. 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 and distinguished elements:

- I, the totally-ordered set of *locations*.

- T, the set of *tags*, defined as $\mathbb{N} \times I$.
- M, the set of *messages*.
- X, the set of *object identifiers*, partitioned into subsets X_i, $i \in I$. X_i is the set of identifiers for objects that may be created at location i. For any $x \in X$, $(i_0)_x$ denotes the unique i such that $x \in X_i$.
- For each $x \in X$:
  - 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 $W \in \mathit{write\text{-}quorums}(c)$, $R \cap W \neq \emptyset$.

We also define:

- 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 $C_\pm$. We extend the update and extend operators elementwise to binary operations on CMap.
- $\mathit{truncate}$, 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 $\perp$.
- $\mathit{Truncated}$, 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 $cm \in \mathit{Usable}$ 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 $cm \in \mathit{Usable}$ 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 $cm(k+1) \in C$.*

2.2 Channel automata

Processes communicate via point to point channels $\mathit{Channel}_{x,i,j}$, one for each $x \in X$, $i, j \in I$ (including the case where $i = j$). $\mathit{Channel}_{x,i,j}$ is accessed using $\mathsf{send}(m)_{x,i,j}$ input actions, by which a sender at location i submits message m associated with object x to the channel, and $\mathsf{receive}(m)_{x,i,j}$ output actions, by which a receiver at location j receives m. When the object x is implicit, we write simply $Channel_{i,j}$ which has actions $\mathsf{send}(m)_{i,j}$ and $\mathsf{receive}(m)_{i,j}$.

Channels may lose, duplicate, and reorder messages, but cannot manufacture new messages. 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. Formally, we model the channel as a multiset. A send adds the message to the multiset, but a receive does not remove the message.

3 Reconfigurable Atomic Memory Service Specification

In this section, we give the 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, where are stated and proved in Sections 10 and 11. The external signature appears in Figure 1.

The client at location i requests to join the system for a particular object x by performing a $\mathsf{join}(rambo, J)_{x,i}$ 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 $\mathsf{join\text{-}ack}(rambo)_{x,i}$ output action.

The client at i initiates a read (resp., write) operation using a read_i (resp., 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 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 report_i output action. Finally, a crash at location i is modelled using a fail_i input action. We do not explicitly model graceful process "leaves," but instead we model process departures as failures.

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

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 on the behavior of the clients:

- *Well-formedness:*
  - For every x and i:
    - No $\mathsf{join}(rambo, *)_{x,i}$, $\mathsf{read}_{x,i}$, $\mathsf{write}(*)_{x,i}$, or $\mathsf{recon}(*, *)_{x,i}$ event is preceded by a fail_i event.
    - At most one $\mathsf{join}(rambo, *)_{x,i}$ event occurs.
    - Any $\mathsf{read}_{x,i}$, $\mathsf{write}(*)_{x,i}$, or $\mathsf{recon}(*, *)_{x,i}$ event is preceded by a $\mathsf{join\text{-}ack}(rambo)_{x,i}$ 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 join-ack$(rambo)_{x,j}$ event for every $j \in members(c')$.

This says participant i can request reconfiguration from c to c' only if i has previously received 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 join-ack$(rambo)_{x,i}$, read-ack$(*)_{x,i}$, write-ack$_{x,i}$, recon-ack$(*)_{x,i}$, or report$(*)_{x,i}$ event is preceded by a fail$_i$ event.
  - Any join-ack$(rambo)_{x,i}$ (resp., read-ack$(*)_{x,i}$, write-ack$_{x,i}$, recon-ack$(*)_{x,i}$) event has a preceding join$(rambo, *)_{x,i}$ (resp., read$_{x,i}$, write$(*)_{x,i}$, 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:
  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 π_1 and π_2 such that π_1 completes before π_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 $(v_0)_x$.

4 Overview

The rest of the paper is devoted to presenting our implementation of RAMBO. The 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 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.

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 Section 5, we present the specification for the $Recon$ service as an external signature and set of traces, postponing its implementation to a later section. Our later analysis of the main *Reader-Writer* algorithm uses the $Recon$ service as a black box, relying on the specification presented herein.

In Section 6, we present the main *Reader-Writer* algorithm. This algorithm is at the heart of the RAMBO protocol, and shows how to perform reads and writes, as well as to garbage-collect old configurations that are no longer needed. In Section 7, we show that the main *Reader-Writer* algorithm satisfies the safety properties described in Section 3.

In Section 8, we present our implementation of $Recon$ and show that it satisfies the specified properties. This concludes are presentation of the RAMBO algorithm.

In Section 9, we (briefly) describe our timed model of computation, and present a series of assumptions regarding failures, the frequency of reconfiguration requests, etc. In Section 10, we present a series of results on latency and fault-tolerance under the assumption that the system is well-behaved. In Section 11, we present similar results for the case where the system may be initially badly behaved, but is *eventually* well-behaved.

[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 [39]. Lemma 13.16 of [39] is presented for a setting 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.

5 Reconfiguration Service Specification

In this section, we present the specification for the $Recon$ service as an external signature and set of traces. We present our implementation of $Recon$ later in Section 8, after we present the main $Reader\text{-}Writer$ algorithm and the proof of its safety properties.

The interface for $Recon$ appears in Figure 2. The client of $Recon$ at location i requests to join the reconfiguration service by performing a $\mathsf{join}(recon)_i$ 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 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 report_i output action. Crashes are modeled using 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 $Reader\text{-}Writer$ 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.

Input:	Output:
$\mathsf{join}(recon)_i$, $i \in I$	$\mathsf{join\text{-}ack}(recon)_i$, $i \in I$
$\mathsf{recon}(c, c')_i$, $c, c' \in C$, $i \in members(c)$	$\mathsf{recon\text{-}ack}(b)_i$, $b \in \{ok, nok\}$, $i \in I$
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

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

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

The safety guarantees provided by the service are as follows:

- *Well-formedness:* For every i:
  - No $\mathsf{join\text{-}ack}(recon)_i$, $\mathsf{recon\text{-}ack}(*)_i$, $\mathsf{report}(*)_i$, or $\mathsf{new\text{-}config}(*, *)_i$ event is preceded by a fail_i event.
  - Any $\mathsf{join\text{-}ack}(recon)_i$ (resp., $\mathsf{recon\text{-}ack}(c)_i$) event has a preceding $\mathsf{join}(recon)_i$ (resp., 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 $\mathsf{recon\text{-}ack}(nok)_{i'}$ 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 $\mathsf{new\text{-}config}(c, k')_{i'}$ both occur, then $k = k'$. (The same configuration identifier cannot be assigned to two different positions in the sequence of configuration identifiers.)

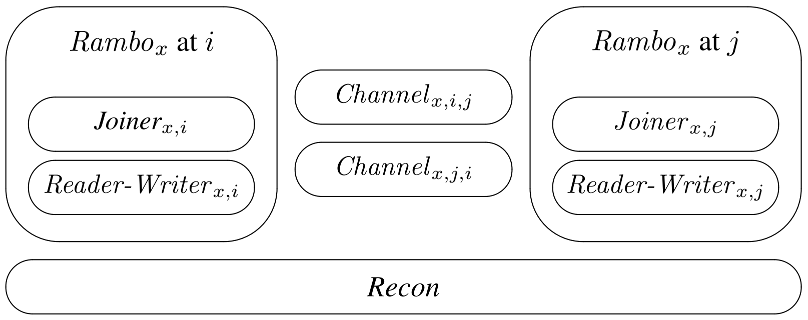

Figure 3: RAMBO architecture: The diagram depicts the *Joiner* and *Reader-Writer* automata at i and j, the *Channel* automata, and the *Recon* service.

6 Implementation of RAMBO

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

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

6.1 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 $\mathsf{join}(rambo, J)$ 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.

6.2 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 a $Reader\text{-}Writer_i$ automaton for each i. The $Reader\text{-}Writer_i$ components interact with the *Recon* service and communicate using point-to-point asynchronous channels.

6.2.1 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 $status = idle$, $Reader\text{-}Writer_i$ does not respond to any inputs (except for join) and does not perform any locally controlled actions. When $status = joining$, $Reader\text{-}Writer_i$ is receptive to inputs

Signature:

```
Input:
  join(rambo, J)_i, J a finite subset of I − {i}
  join-ack(r)_i, r ∈ {recon, rw}
  fail_i

Output:
  send(join)_{i,j}, j ∈ I − {i}
  join(r)_i, r ∈ {recon, rw}
  join-ack(rambo)_i
```

State:

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

Transitions:

```
Input join(rambo, J)_i
Effect:
    if ¬failed then
      if status = idle then
        status ← joining
        hints ← J

Output send(join)_{i,j}
Precondition:
    ¬failed
    status = joining
    j ∈ hints
Effect:
    none

Output join(r)_i
Precondition:
    ¬failed
    status = joining
    child-status(r) = idle
Effect:
    child-status(r) ← joining

Input join-ack(r)_i
Effect:
    if ¬failed then
      if status = joining then
        child-status(r) ← active

Output join-ack(rambo)_i
Precondition:
    ¬failed
    status = joining
    ∀r ∈ {recon, rw}:
      child-status(r) = active
Effect:
    status ← active

Input fail_i
Effect:
    failed ← true
```

Figure 4: $Joiner_i$

but still does not perform any locally controlled actions. When $status = active$, 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 $Reader\text{-}Writer$ 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 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 $\bot$.

The $pnum1$ variable and $pnum2$ array are used to implement a handshake that identifies "recent" messages. $Reader\text{-}Writer_i$ 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

Signature:

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

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

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:

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

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

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

Figure 5: $\mathit{Reader\text{-}Writer}_i$: Signature and state

propagate phase, as described below.) For every j, including $j = i$, $\mathit{Reader\text{-}Writer}_i$ uses $\mathit{pnum2}(j)$ 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.

6.2.2 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 $\mathsf{join}(\mathit{rw})_i$ input occurs when $\mathit{status} = \mathit{idle}$, if i is the object's creator i_0, then status immediately becomes active, which means that $\mathit{Reader\text{-}Writer}_i$ is ready for full participation in the protocol. Otherwise, status becomes $\mathit{joining}$, which means that $\mathit{Reader\text{-}Writer}_i$ is receptive to inputs but not ready to perform any locally controlled actions. In either case, $\mathit{Reader\text{-}Writer}_i$ records itself as a member of its own world. From this point on, $\mathit{Reader\text{-}Writer}_i$ 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 $\mathit{Reader\text{-}Writer}$ automata.)

If $\mathit{status} = \mathit{joining}$, then status becomes active when $\mathit{Reader\text{-}Writer}_i$ 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 $\neg failed$ then
    if $status = idle$ then
      if $i = i_0$ then
        $status \leftarrow active$
      else
        $status \leftarrow joining$
      $world \leftarrow world \cup \{i\}$

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

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

Input fail$_i$
Effect:
  $failed \leftarrow true$
```

Figure 6: $Reader\text{-}Writer_i$: 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 $Reader\text{-}Writer_i$ 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, $Reader\text{-}Writer_i$ sets its *tag* to t and its *value* to the incoming value v. $Reader\text{-}Writer_i$ also updates its own configuration map, *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 $c \in C$, process i sets its $cmap(k)$ to c. Also, if $cmap(k)$ is either $\bot$ or a configuration identifier in C, and $cm(k) = \pm$, indicating that the sender knows that configuration k has already been garbage-collected, then $Reader\text{-}Writer_i$ sets its $cmap(k)$ to $\pm$. $Reader\text{-}Writer_i$ 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.

When $Reader\text{-}Writer_i$ is conducting a phase of a read, write, or garbage-collection operation, it verifies that the incoming message is "recent", in the sense that the sender j sent it after j received a message from i that was sent after i began the current phase. $Reader\text{-}Writer_i$ 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, $Reader\text{-}Writer_i$ 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 $Reader\text{-}Writer_i$ 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, $Reader\text{-}Writer_i$

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

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

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

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

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

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

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

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

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

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

chooses the $CMap$ $truncate(cmap)$, which is defined to include all the configuration identifiers in the local $cmap$ up to the first $\perp$. When new $CMap$ information arrives during the phase, $op.cmap$ is "extended" by adding all newly-discovered configuration identifiers, up to the first $\perp$ in the incoming $CMap$ 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 $\mathit{Truncated}$), then $\mathit{Reader\text{-}Writer}_i$ can infer that it has been conducting the operation 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 $\mathit{truncate}(\mathit{cmap})$ for the latest local cmap.

In between restarts, while process i is engaged in a single attempt to complete a phase, no configuration identifier is ever removed from $\mathit{op.cmap}$, 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 $\mathit{op.cmap}$, 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 $\mathit{op.cmap}$. Let v and t denote process i’s value and 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 v 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” processes have acquired tags that are at least as great as t (and associated values). Again, the information is propagated in the background, and $\mathit{op.cmap}$ is managed as described above. The propagation phase ends once a “propagation fixed point” is reached, when process i has received recent responses from some write-quorum of each configuration in the current $\mathit{op.cmap}$. When this occurs, we know that the tag value of each process in each of these write-quorums is at least as great as t.

Processing for a write operation, say $\mathsf{write}(v)_i$, for a particular i and v, is similar to that for a read 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 $\mathit{Reader\text{-}Writer}_i$ defines the tag for its write operation to be the pair $(n+1, i)$. $\mathit{Reader\text{-}Writer}_i$ 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 $\mathit{op.cmap}$ 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., [8, 20, 40]). 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 $\mathit{Reader\text{-}Writer}_i$ hears 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. The configuration identifiers used in performing query and propagation phases of reads and writes are those in $\mathit{truncate}(\mathit{cmap})$, that is, all configurations that have not been garbage-collected and that appear before the first $\perp$.

There are two situations in which $\mathit{Reader\text{-}Writer}_i$ may garbage-collect a configuration identifier, say, the one in $\mathit{cmap}(k)$. First, $\mathit{Reader\text{-}Writer}_i$ can garbage-collect $\mathit{cmap}(k)$ if it ever hears that another process has already garbage-collected it. This happens when a $\mathsf{recv}_{*,i}$ event occurs in which $\mathit{cm}(k) = \pm$. The second, more interesting situation is where $\mathit{Reader\text{-}Writer}_i$ acquires enough information to garbage-collect configuration k on its own. $\mathit{Reader\text{-}Writer}_i$ 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. $\mathit{Reader\text{-}Writer}_i$ may initiate a garbage-collection of all configurations with index $\leq k$ when both $\mathit{cmap}(k) \in C$ and $\mathit{cmap}(k+1) \in C$, and when for every index $\ell \leq k$, $\mathit{cmap}(\ell) \neq \perp$. Garbage-collection operations may proceed concurrently with read or write operations at the same node.

In the query phase of a garbage-collection operation, process i communicates with both a read-quorum and a write-quorum of every active configuration with index $\leq k$, that is, every configuration with index $\ell \leq k$ such that $gc.cmap(\ell) \in C$. The query phase accomplishes two tasks: First, $\mathit{Reader\text{-}Writer}_i$ ensures that certain information is conveyed to the processes in the read- and write-quorums of all active configurations with index $\leq k$. In particular, all these processes learn about configuration $k+1$, and also learn that all configurations smaller than $k+1$ are being 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 at a later time by someone who is trying to access a quorum of configuration k, j is able to tell that process about the existence of the later configuration k. Second, in the query phase, $\mathit{Reader\text{-}Writer}_i$ collects tag and $value$ information from the read- and write-quorums that it accesses. This ensures that, by the end of the query phase, $\mathit{Reader\text{-}Writer}_i$'s tag is equal to some value t that is at least as great as the tag that each of the quorum members had when it sent a message to $\mathit{Reader\text{-}Writer}_i$ for the query phase. In the propagation phase, $\mathit{Reader\text{-}Writer}_i$ ensures that a write-quorum of the new configuration, $k+1$, have acquired tags that are at least as great as t.

Note that the two phases of garbage-collection differ from the two phases of the read and write operations in that they do not involve "fixed point" tests. In this case, $\mathit{Reader\text{-}Writer}_i$ does not start with some set of configurations in $op.cmap$ and extend it as it learns about more configurations. Rather, $\mathit{Reader\text{-}Writer}_i$ knows ahead of time which configurations are being used—those that are active in $gc.cmap$—and uses only quorums from those configurations.

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

```
Internal gc($k$)$_i$
Precondition:
    $\neg failed$
    $status = active$
    $gc.phase = idle$
    $cmap(k) \in C$
    $cmap(k-1) \in C$
    $\forall \ell < k : cmap(\ell) \neq \bot$
Effect:
    $pnum1 \leftarrow pnum1 + 1$
    $gc.phase \leftarrow query$
    $gc.pnum \leftarrow pnum1$
    $gc.cmap \leftarrow truncate(cmap)$
    $gc.acc \leftarrow \emptyset$
    $gc.target \leftarrow k$

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

Internal gc-prop-fix($k$)$_i$
Precondition:
    $\neg failed$
    $status = active$
    $gc.phase = prop$
    $gc.target = k$
    $\exists W \in write\text{-}quorums(gc.cmap(k)) : W \subseteq gc.acc$
Effect:
    for $\ell < k$ do
      $cmap(\ell) \leftarrow \pm$

Internal gc-ack($k$)$_i$
Precondition:
    $\neg failed$
    $status = active$
    $gc.target = k$
    $\forall \ell < k : \ cmap(\ell) = \pm$
Effect:
    $gc.phase = idle$
```

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

6.3 The complete algorithm

The complete implementation $\mathcal{S}$ is the composition of all the automata defined above—the $Joiner_i$ and $Reader\text{-}Writer_i$ 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.

7 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 7.1 *Let β be a trace of the system $\mathcal{S}$. If β satisfy the* RAMBO *environment assumptions, then β satisfies the* RAMBO *service guarantees (well-formedness and atomicity).*

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 β 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 β satisfying the conditions listed in the atomicity definition in Section 3.

The proof is carried out in several stages. First, in Section 7.1, we establish some notational conventions and define some useful history variables. In Sections 7.2 and 7.3, we present some basic invariants and guarantees. The following two subsections describe information propagation between operations: in Section 7.4 we establish a relationship between garbage-collection operations; in Section 7.5 we show how information is propagated from read/write operations to garbage-collection operations, and *vice versa*. In Section 7.6, we consider the relationship between two read or write operations, culminating in Lemma 7.17, which says that tags are monotonic with respect to non-concurrent read or write operations. Finally, Section 7.7 uses the tags to define a partial order on operations and verifies the four properties required for atomicity.

Throughout this section, we consider executions of $\mathcal{S}$ whose trace satisfies 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 $\mathcal{S}$.

7.1 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* is a pair (n, i) consisting of a natural number n and an index $i \in I$. Here, i is the index of the process running the operation, and n is the value of $pnum1_i$ just after the read, write, or gc event of the operation occurs. We introduce the following history variables:

- $in\text{-}transit$, a set of messages, initially $\emptyset$.
  A message is added to the set when it is sent by any $Reader\text{-}Writer_i$ to any $Reader\text{-}Writer_j$. No message is ever removed from this set.
- For every $k \in \mathbb{N}$:
  1. $c(k) \in C$, initially undefined.
     This is set when the first new-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 π:
  1. $tag(\pi) \in T$, initially undefined.
     This is set to the value of tag at the process running π, at the point right after π's query-fix or gc-query-fix event occurs. If π is a read or garbage-collection operation, this is the highest tag that

it encounters during the query phase. If π is a write operation, this is the new tag that is selected for performing the write.

- For every read or write operation π:

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

- For every garbage-collection operation γ:

  1. $removal\text{-}set(\gamma)$, a subset of $\mathbb{N}$, initially undefined.
     This is set in the gc(k) step of γ, to the set $\{\ell : \ell < k, cmap(\ell) \neq \pm\}$ in the pre-state.
  2. $target(\gamma)$, a configuration, initially undefined.
     This is set in the gc(k) step of γ to k.
  3. $R(\gamma, \ell)$, for $\ell \in \mathbb{N}$, a subset of I, initially undefined.
     This is set in the gc-query-fix step of γ, for each $\ell \in removal\text{-}set(\gamma)$, to an arbitrary read-quorum $R \in read\text{-}quorums(c(\ell))$ such that $R \subseteq gc.acc$ in the pre-state.
  4. $W_1(\gamma, \ell)$, for $\ell \in \mathbb{N}$, a subset of I, initially undefined.
     This is set in the gc-query-fix step of γ, for each $\ell \in removal\text{-}set(\gamma)$, to an arbitrary write-quorum $W \in write\text{-}quorums(c(\ell))$ such that $W \subseteq gc.acc$ in the pre-state.
  5. $W_2(\gamma)$, a subset of I, initially undefined.
     This is set in the gc-prop-fix step of γ, to an arbitrary $W \in write\text{-}quorums(c(k))$ such that $W \subseteq gc.acc$ in the pre-state.

In any good execution α, we define the following events (more precisely, we are giving additional names to some existing events):

1. For every read or write operation π:

   (a) query-phase-start(π), initially undefined.
   This is defined in the query-fix step of π, to be the unique earlier event at which the collection of query results was started and not subsequently restarted. This is either a read, write, or recv event.

   (b) prop-phase-start(π), initially undefined.
   This is defined in the prop-fix step of π, 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.

7.2 Configuration map invariants

In this section, we give invariants describing the kinds of configuration maps that may appear in various places in the state of $\mathcal{S}$. We begin with a lemma saying that various operations yield or preserve the "usable" property:

Lemma 7.2 *(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. Immediate, by the definition of $update$, $extend$, and $truncate$. $\Box$

The next invariant (recall that 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 $gc.target_i = k$, then: (1) $\forall \ell : \ell \leq k \Rightarrow cmap(\ell)_i \in C \cup \{\pm\}$. (2) If $k_1 = \min\{\ell : \ell \leq k$ and $gc.cmap(\ell) \neq \pm\}$ then $k_1 = 0$ or $cmap(k_1 - 1)_i = \pm$.*

Proof. The precondition of $\mathsf{gc}(k)_i$ ensures that this property holds when a garbage-collection begins, and the monotonicity of updates to the $cmap_i$ ensures that it is maintained. $\square$

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 $i \in I$; $op.cmap_i$ for some $i \in I$ for which $op.phase \neq idle$; $query\text{-}cmap(\pi)$ or $prop\text{-}cmap(\pi)$ for any operation π; $gc.cmap_i$ for some $i \in I$ for which $gc.phase \neq idle$. Then $cm \in Usable$.*

Proof. By induction on the length of a finite good execution: it is easily observed that no action causes a $CMap$ to become unusable. $\square$

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 $op.phase_i \neq idle$, or as $query\text{-}cmap(\pi)$ or $prop\text{-}cmap(\pi)$ for any operation π. Then: (1) $cm \in Truncated$, and (2) cm consists of finitely many $\pm$ entries followed by finitely many C entries followed by an infinite number of $\perp$ entries.*

Proof. The claim for $op.cmap_i$ with respect to Part 1 follows by induction: during read_i, write_i or $\mathsf{query\text{-}fix}_i$ events, $op.cmap_i$ is set to $truncate(cmap_i)$, i.e., to a value in $Truncated$. Every recv_i event maintains this. The same properties for $query\text{-}cmap_i$ and $prop\text{-}cmap_i$ follow by definition. Part 2 holds by the fact that $cm \in Usable$, as per Invariant 2. $\square$

7.3 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 read and write operations, and during the two phases of garbage collection. Specifically, each of these lemmas asserts that when some node i completes a phase, then for every node j in some quorum (or pair of quorums), there is some pair of messages m and m' such that:

1. m is sent from i to j after the phase begins.
2. m' is sent from j to i after j receives m.
3. m' is received by i before the end of the phase.
4. If the tag is a query phase, then the tag of the operation or garbage collection is at least as large as the tag of j when message m' is sent. If the phase is a propagate phase, then the tag of j is at least as large as the tag of the operation or garbage collection.

Additionally, these lemmas make claims about the $cmap$ associated with the operation or garbage collection.

Note that these lemmas treat the case where $j = i$ uniformly with the case where $j \neq i$. This is because, in the $Reader\text{-}Writer$ algorithm, communication from a location to itself is treated uniformly with communication between two different locations.

We first consider the query phase of read and write operations:

Lemma 7.3 *Suppose that a* query-fix$_i$ *event for a read or write operation* π *occurs in an execution* α. *Let* $k, k' \in \mathbb{N}$. *Suppose* $\mathit{query\text{-}cmap}(\pi)(k) \in C$ *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(π) *event.*
2. m' *is sent after* j *receives* m.
3. m' *is received before the* query-fix *event of* π.
4. *If* t *is the value of* tag_j *in any state before* j *sends* m', *then:*
   (a) $\mathit{tag}(\pi) \geq t$.
   (b) *If* π *is a write operation then* $\mathit{tag}(\pi) > t$.
5. *If* $\mathit{cmap}(\ell)_j \neq \perp$ *for all* $\ell \leq k'$ *in any state before* j *sends* m', *then* $\mathit{query\text{-}cmap}(\pi)(\ell) \in C$ *for some* $\ell \geq k'$.

Proof. The phase number discipline implies the existence of the claimed messages m and m'. It is then easy to see that the tag component of message m' is $\geq t$, ensuring that $\mathit{tag}(\pi) \geq t$, or, in the case of a write operation, $\mathit{tag}(\pi) > t$.

Next, assume that $\mathit{cmap}(\ell)_j \neq \perp$ for all $\ell \leq k'$ prior to j sending message m'. Since i receives m' after the query-phase-start(π) event, we can conclude that after receiving m', $\mathit{truncate}(\mathit{op.cmap}_i)(\ell) \neq \perp$ for all $\ell \leq k'$. Since $\mathit{op.cmap}_i$ is Usable, as per Invariant 2, we conclude that $\mathit{op.cmap}_i(\ell) \in C$ for some $\ell \geq k'$, implying the desired claim. □

Next, we consider the propagation phase of read and write operations:

Lemma 7.4 *Suppose that a* prop-fix$_i$ *event for a read or write operation* π *occurs in an execution* α. *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(π) *event.*
2. m' *is sent after* j *receives* m.
3. m' *is received before the* prop-fix *event of* π.
4. *In any state after* j *receives* m, $\mathit{tag}_j \geq \mathit{tag}(\pi)$.
5. *If* $\mathit{cmap}(\ell)_j \neq \perp$ *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'. It is then easy to see that tag component of message m is $\geq \mathit{tag}(\pi)$, ensuring that after j receives message m, $\mathit{tag}_j \leq \mathit{tag}(\pi)$. The final conclusion is identical to Lemma 7.3, with respect to the $\mathit{prop\text{-}cmap}(\pi)$ rather than the $\mathit{query\text{-}cmap}(\pi)$. □

In the following two lemmas, we consider the behavior of the two phases of a garbage-collection operation. We begin with the query phase:

Lemma 7.5 *Suppose that a* gc-query-fix$(k)_i$ *event for garbage-collection operation* γ *occurs in an execution* α *and* $k' \in \mathit{removal\text{-}set}(\gamma)$. *Suppose* $j \in R(\gamma, k') \cup W_1(\gamma, k')$. *Then there exist messages* m *from* i *to* j *and* m' *from* j *to* i *such that:*

1. m *is sent after the* gc$(k)_i$ *event of* γ.
2. m' *is sent after* j *receives* m.
3. m' *is received before the* gc-query-fix$(k)_i$ *event of* γ.
4. *If* t *is the value of* tag_j *in any state before* j *sends* m', *then* $\mathit{tag}(\gamma) \geq t$.
5. *In any state after* j *receives* m, $\mathit{cmap}(\ell)_j \neq \perp$ *for all* $\ell \leq k$.

Proof. The phase number discipline implies the existence of the claimed messages m and m'. It is then easy to see that the tag component of message m' is $\geq t$, ensuring that $\mathit{tag}(\gamma) \geq t$.

The final claim holds since, when the gc$(k)_i$ event occurs, we know that $\mathit{cmap}(\ell)_i \neq \perp$ for all $\ell \leq k$ according to the precondition. Thus the same property holds for the cm component of message m, and hence for j after receiving message m. □

Finally, we consider the propagation phase of a garbage-collection operation:

Lemma 7.6 *Suppose that a* $\mathsf{gc\text{-}prop\text{-}fix}(k)_i$ *event for a garbage-collection operation* γ *occurs in an execution* α*. 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* γ*.*
2. *m' is sent after j receives m.*
3. *m' is received before the* $\mathsf{gc\text{-}prop\text{-}fix}(k)_i$ *event of* γ*.*
4. *In any state after j receives m,* $\mathit{tag}_j \geq \mathit{tag}(\gamma)$*.*

Proof. The phase number discipline implies the existence of the claimed messages m and m'. It is then easy to see that the tag component of message m is $\geq \mathit{tag}(\gamma)$, ensuring that after j receives message m, $\mathit{tag}_j \geq \mathit{tag}(\gamma)$. □

7.4 Garbage collection

This section establishes lemmas describing information flow between garbage-collection operations. The key result in this section is Lemma 7.9, which asserts the existence of a sequence of garbage-collection operations $\gamma_0, \ldots, \gamma_k$ which have certain key properties. In particular, the sequence of garbage-collection operations removes all the configurations installed in an execution, except for the last, and the tags associated with the garbage-collection operations are monotonically non-decreasing, guaranteeing that value/tag information is propagated to newer configurations.

We say that a sequence of garbage-collection operations $\gamma_\ell, \ldots, \gamma_k$ in some execution α is an (ℓ, k)*-gc-sequence* if it satisfies the following three properties:

1. $\forall\ s : \ell \leq s \leq k,\ s \in \mathit{removal\text{-}set}(\gamma_s)$,
2. $\forall\ s : \ell \leq s < k$, if $\gamma_s \neq \gamma_{s+1}$, then the $\mathsf{gc\text{-}prop\text{-}fix}$ event of γ_s occurs in α and precedes the gc event of γ_{s+1}, and
3. $\forall\ s : \ell \leq s < k$, if $\gamma_s \neq \gamma_{s+1}$, then $\mathit{target}(\gamma_s) \in \mathit{removal\text{-}set}(\gamma_{s+1})$.

Notice that an (ℓ, k)-gc-sequence may well contain the same garbage-collection operation multiple times. If two elements in the sequence are distinct operations, then the earlier operation in the sequence completes before the later operation is initiated. Also, the target of an operation in the sequence is removed by the next distinct operation in the sequence. These properties imply that the garbage-collection process obeys a sequential discipline.

We begin by showing that if there is no garbage-collection operation with target k, then configurations with index $k-1$ and k are always removed together.

Lemma 7.7 *Suppose that $k > 0$, and α is an execution in which no* $\mathsf{gc\text{-}prop\text{-}fix}(k)$ *event occurs in α. Suppose that cm is a CMap that appears as one of the following, for some state in α:*

1. *The cm component of some message in $\mathit{in\text{-}transit}$.*
2. *cmap_i, for any $i \in I$.*
3. *The $\mathit{op}.\mathit{cmap}_i$, for any $i \in I$.*
4. *The $\mathit{gc}.\mathit{cmap}_i$, for any $i \in I$.*

If $cm(k-1) = \pm$ then $cm(k) = \pm$.

Proof. The proof follows by induction on events in α. The base case is trivially true. In the inductive step, notice that the only event that can set a CMap $cm(k-1) = \pm$ without also setting $cm(k) = \pm$ is a $\mathsf{gc\text{-}prop\text{-}fix}(k)$ event, which we have assumed does not occur in α. □

The following corollary says that if a $\mathsf{gc}(k)$ event occurs in α and k' is the smallest configuration in the removal set, then there is some garbage collection γ' that completes before the $\mathsf{gc}(k)$ event with target k'.

Corollary 7.8 *Assume that a* $\mathsf{gc}(k)_i$ *event occurs in an execution* α*, associated with garbage collection* γ*. Let* $k' = \min\{\mathit{removal\text{-}set}(\gamma)\}$*, and assume* $k' > 0$*. Then for some* j*, a* $\mathsf{gc\text{-}prop\text{-}fix}(k')_j$ *event occurs in* α *and precedes the* $\mathsf{gc}(k)_i$ *event.*

Proof. Immediately prior to the gc event, $\mathit{cmap}(k'-1)_i = \pm$ and $\mathit{cmap}(k')_i \neq \pm$. Lemma 7.7 implies that some $\mathsf{gc\text{-}prop\text{-}fix}(k')$ event for some operation γ' occurs in α, and this event necessarily precedes the gc event. □

The next lemma says that if some garbage-collection operation γ removes a configuration with index k in an execution α, then there exists a $(0, k)$-gc-sequence of garbage-collection operations. The lemma constructs such a sequence: for every configuration with an index smaller than k, it identifies a single garbage-collection operation that removes that configuration, and adds it to the sequence.

Lemma 7.9 *If a* gc_i *event for garbage-collection operation* γ *occurs in an execution* α *such that* $k \in \mathit{removal\text{-}set}(\gamma)$*, then there exists a* $(0, k)$*-gc-sequence (possibly containing repeated elements) of garbage-collection operations.*

Proof. We construct the sequence in reverse order, first defining γ_k, and then at each step defining the preceding element. We prove the lemma by backward induction on ℓ, for $\ell = k$ down to $\ell = 0$, maintaining the property that $\gamma_\ell, \ldots, \gamma_k$ is an (ℓ, k)-gc-sequence. To begin the induction, we define $\gamma_k = \gamma$, satisfying the property that $k \in \mathit{removal\text{-}set}(\gamma_k)$; the other two properties are vacuously true.

For the inductive step, we assume that γ_ℓ has been defined and that $\gamma_\ell, \ldots, \gamma_k$ is an (ℓ, k)-gc-sequence. If $\ell = 0$, then γ_0 has been defined, and we are done. Otherwise, we need to define $\gamma_{\ell-1}$. If $\ell - 1 \in \mathit{removal\text{-}set}(\gamma_\ell)$, then let $\gamma_{\ell-1} = \gamma_\ell$, and all the necessary properties still hold.

Otherwise, $\ell - 1 \notin \mathit{removal\text{-}set}(\gamma_\ell)$ and $\ell \in \mathit{removal\text{-}set}(\gamma_\ell)$, which implies that $\ell = \min\{\mathit{removal\text{-}set}(\gamma_\ell)\}$. By Corollary 7.8, there occurs in α a garbage-collection operation that we label $\gamma_{\ell-1}$ with the following properties: (i) the gc-prop-fix event of $\gamma_{\ell-1}$ precedes the gc event of γ_ℓ, and (ii) $\mathit{target}(\gamma_{\ell-1}) = \min\{k' : k' \in \mathit{removal\text{-}set}(\gamma_\ell)\}$, i.e., $\mathit{target}(\gamma_{\ell-1}) = \ell$.

Since $\mathit{removal\text{-}set}(\gamma_{\ell-1}) \neq \emptyset$, by definition and the precondition of a gc event, this implies that $\ell - 1 \in \mathit{removal\text{-}set}(\gamma_{\ell-1})$, proving Property 1 of the $(\ell - 1, k)$-gc-sequence definition. Property 2 and Property 3 follow similarly from the choice of $\gamma_{\ell-1}$. □

The sequential nature of garbage collection has a nice consequence for propagation of tags: for any (ℓ, k)-gc-sequence of garbage-collection operations, $\mathit{tag}(\gamma_s)$ is nondecreasing in s.

Lemma 7.10 *Let* $\gamma_\ell, \ldots, \gamma_k$ *be an* (ℓ, k)*-gc-sequence of garbage-collection operations. Then* $\forall\ s : \ell \leq s < k,\ \mathit{tag}(\gamma_s) \leq \mathit{tag}(\gamma_{s+1})$.

Proof. If $\gamma_s = \gamma_{s+1}$, then the claim follows trivially. Therefore assume that $\gamma_s \neq \gamma_{s+1}$; this implies that the gc-prop-fix event of γ_s precedes the gc event of γ_{s+1}. Let k_2 be the target of γ_s. We know by assumption that $k_2 \in \mathit{removal\text{-}set}(\gamma_{s+1})$. Therefore, $W_2(\gamma_s)$, a write-quorum of configuration $c(k_2)$, has at least one element in common with $R(\gamma_{s+1}, k_2)$; label this node j. By Lemma 7.6, and the monotonicity of tag_j, after the gc-prop-fix event of γ_s we know that $\mathit{tag}_j \geq \mathit{tag}(\gamma_s)$. Then by Lemma 7.5 $\mathit{tag}(\gamma_{s+1}) \geq \mathit{tag}_j$. Therefore $\mathit{tag}(\gamma_s) \leq \mathit{tag}(\gamma_{s+1})$. □

Corollary 7.11 *Let* $\gamma_\ell, \ldots, \gamma_k$ *be an* (ℓ, k)*-gc-sequence of garbage-collection operations. Then* $\forall\ s, s' : 0 \leq s \leq s' \leq k$*,* $\mathit{tag}(\gamma_s) \leq \mathit{tag}(\gamma_{s'})$

Proof. This follows immediately from Lemma 7.10 by induction. □

7.5 Behavior of a read or a write following a garbage collection

Now we describe the relationship between a read or write operation and a preceding garbage-collection operation. The key result in this section, Lemma 7.14, shows that if a garbage-collection operation completes prior to some read or write operation, then the tag of the read or write operation is no smaller than the tag of the garbage collection.

The first lemma shows that if, for some read or write operation, k is the smallest index such that $\textit{query-cmap}(k) \in C$, then some garbage-collection operation with target k precedes the read or write operation.

Lemma 7.12 *Let π be a read or write operation whose* query-fix *event occurs in an execution α. Let k be the smallest element such that $\textit{query-cmap}(\pi)(k) \in C$. Assume $k > 0$. Then there exists a garbage-collection operation γ such that $k = \textit{target}(\gamma)$, and the* gc-prop-fix *event of γ precedes the* query-phase-start(π) *event.*

Proof. This follows immediately from (the contrapositive of) Lemma 7.7. □

Second, if a garbage collection that removes k completes before the query-phase-start event of a read or write operation, then some configuration with index $\geq k+1$ must be included in the $\textit{query-cmap}$ of the later read or write operation. (Otherwise, the read or write operation would have no extant configurations available to it.)

Lemma 7.13 *Let γ be a garbage-collection operation such that $k \in \textit{removal-set}(\gamma)$. Let π be a read or write operation whose* query-fix *event occurs in an execution α. Suppose that the* gc-prop-fix *event of γ precedes the* query-phase-start(π) *event in α. Then $\textit{query-cmap}(\pi)(\ell) \in C$ for some $\ell \geq k+1$.*

Proof. Suppose for the sake of contradiction that $\textit{query-cmap}(\pi)(\ell) \notin C$ for all $\ell \geq k+1$. Fix $k' = \max(\{\ell' : \textit{query-cmap}(\pi)(\ell') \in C\})$. Then $k' \leq k$.

Let $\gamma_0, \ldots, \gamma_k$ be a $(0,k)$-gc-sequence of garbage-collection operations whose existence is asserted by Lemma 7.9, where $\gamma_k = \gamma$. Then, $k' \in \textit{removal-set}(\gamma_{k'})$, and the gc-prop-fix event of $\gamma_{k'}$ does not come after the gc-prop-fix event of γ in α, and hence precedes the query-phase-start(π) event in α.

Since $k' \in \textit{removal-set}(\gamma_{k'})$, write-quorum $W_1(\gamma_{k'}, k')$ is defined. Since $\textit{query-cmap}(k') \in C$, the read-quorum $R(\pi, k')$ is defined. Choose $j \in W_1(\gamma_{k'}, k') \cap R(\pi, k')$. Assume that $k_t = \textit{target}(\gamma_{k'})$. Notice that $k' < k_t$. Then Lemma 7.5 and monotonicity of $\textit{cmap}$ imply that, in the state just prior to the gc-query-fix event of $\gamma_{k'}$, $\textit{cmap}(\ell)_j \neq \perp$ for all $\ell \leq k_t$. Then Lemma 7.3 implies that $\textit{query-cmap}(\pi)(\ell) \in C$ for some $\ell \geq k_t$. But this contradicts the choice of k'. □

Finally, we show that the $\textit{tag}$ is correctly propagated from a garbage-collection operation to a following read or write operation. For this lemma, we assume that $\textit{query-cmap}(k) \in C$, where k is the target of the garbage collection.

Lemma 7.14 *Let γ be a garbage-collection operation, and assume that $k = \textit{target}(\gamma)$. Let π be a read or write operation whose* query-fix *event occurs in an execution α. Suppose that the* gc-prop-fix *event of γ precedes the* query-phase-start(π) *event in execution α. Suppose also that $\textit{query-cmap}(\pi)(k) \in C$. Then:*

1. *$\textit{tag}(\gamma) \leq \textit{tag}(\pi)$.*
2. *If π is a write operation then $\textit{tag}(\gamma) < \textit{tag}(\pi)$.*

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

Lemma 7.6 implies that, in any state after the gc-prop-fix event for γ, $\textit{tag}_j \geq \textit{tag}(\gamma)$. Since the gc-prop-fix event of γ precedes the query-phase-start(π) event, we have that $t \geq \textit{tag}(\gamma)$, where t is defined to be the value of $\textit{tag}_j$ just before the query-phase-start(π) event. Then Lemma 7.3 implies that $\textit{tag}(\pi) \geq t$, and if π is a write operation, then $\textit{tag}(\pi) > t$. Combining the inequalities yields both conclusions of the lemma. □

7.6 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 now prove some relationships between their $\mathit{query\text{-}cmap}$s, $\mathit{prop\text{-}cmap}$s, and tags. The first lemma says that, when two read or write operations execute sequentially, the smallest configuration index used in the propagation phase of the first operation is no greater than 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 7.15 *Assume π_1 and π_2 are two read or write operations, such that: (1) The* prop-fix *event of π_1 occurs in an execution α. (2) The* query-fix *event of π_2 occurs in α. (3) The* prop-fix *event of π_1 precedes the* query-phase-start(π_2) *event. Then* $\min(\{\ell : \mathit{prop\text{-}cmap}(\pi_1)(\ell) \in C\}) \leq \max(\{\ell : \mathit{query\text{-}cmap}(\pi_2)(\ell) \in C\})$.

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

This implies that some gc-prop-fix event for some garbage-collection operation γ such that $k \in \mathit{removal\text{-}set}(\gamma)$ occurs prior to the prop-fix of π_1, and hence prior to the query-phase-start(π_2) event. Lemma 7.13 then implies that $\mathit{query\text{-}cmap}(\pi_2)(\ell) \in C$ 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 7.16 *Assume π_1 and π_2 are two read or write operations, and $k \in \mathbb{N}$, such that: (1) The* prop-fix *event of π_1 occurs in an execution α. (2) The* query-fix *event of π_2 occurs in α. (3) The* prop-fix *event of π_1 precedes the* query-phase-start(π_2) *event. (4) $\mathit{prop\text{-}cmap}(\pi_1)(k)$ and $\mathit{query\text{-}cmap}(\pi_2)(k)$ are both in C. Then:*

1. $\mathit{tag}(\pi_1) \leq \mathit{tag}(\pi_2)$.
2. *If π_2 is a write then* $\mathit{tag}(\pi_1) < \mathit{tag}(\pi_2)$.

Proof. The hypotheses imply that $\mathit{prop\text{-}cmap}(\pi_1)(k) = \mathit{query\text{-}cmap}(\pi_2)(k) = c(k)$. Then $W(\pi_1, k)$ and $R(\pi_2, k)$ are both defined in α. 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 7.4 implies that, in any state after the prop-fix event of π_1, $\mathit{tag}_j \geq \mathit{tag}(\pi_1)$. Since the prop-fix event of π_1 precedes the query-phase-start(π_2) event, we have that $t \geq \mathit{tag}(\pi_1)$, where t is defined to be the value of tag_j just before the query-phase-start(π_2) event. Then Lemma 7.3 implies that $\mathit{tag}(\pi_2) \geq t$, and if π_2 is a write operation, then $\mathit{tag}(\pi_2) > t$. Combining the inequalities yields both conclusions. ☐

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 7.17 *Assume π_1 and π_2 are two read or write operations, such that: (1) The* prop-fix *of π_1 occurs in an execution α. (2) The* query-fix *of π_2 occurs in α. (3) The* prop-fix *event of π_1 precedes the* query-phase-start(π_2) *event. Then:*

1. $\mathit{tag}(\pi_1) \leq \mathit{tag}(\pi_2)$.
2. *If π_2 is a write then* $\mathit{tag}(\pi_1) < \mathit{tag}(\pi_2)$.

Proof. Let i_1 and i_2 be the indices of the processes that run operations π_1 and π_2, respectively. Let $cm_1 = \mathit{prop\text{-}cmap}(\pi_1)$ and $cm_2 = \mathit{query\text{-}cmap}(\pi_2)$. If there exists k such that $cm_1(k) \in C$ and $cm_2(k) \in C$, then Lemma 7.16 implies the conclusions of the lemma. So from now on, we assume that no such k exists.

Lemma 7.15 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 $s_1 < s_2$, where s_1 is defined to be $\max(\{\ell : cm_1(\ell) \in C\})$ and s_2 is defined to be $\min(\{\ell : cm_2(\ell) \in C\})$.

Lemma 7.12 implies that there exists a garbage-collection operation that we will call γ_{s_2-1} such that $s_2 = \mathit{target}(\gamma_{s_2-1})$, and the gc-prop-fix of γ_{s_2-1} precedes the query-phase-start(π_2) event. Then by Lemma 7.14, $\mathit{tag}(\gamma_{s_2-1}) \leq \mathit{tag}(\pi_2)$, and if π_2 is a write operation then $\mathit{tag}(\gamma_{s_2-1}) < \mathit{tag}(\pi_2)$.

Next we will demonstrate a chain of garbage-collection operations with non-decreasing tags. Lemma 7.9, in conjunction with the already defined γ_{s_2-1}, implies the existence of a $(0, s_2 - 1)$-gc-sequence of garbage-collection operations $\gamma_0, \ldots, \gamma_{s_2-1}$. Since $s_1 \leq s_2 - 1$, we know that $s_1 \in \mathit{removal\text{-}set}(\gamma_{s_1})$. Then Corollary 7.11 implies that $\mathit{tag}(\gamma_{s_1}) \leq \mathit{tag}(\gamma_{s_2-1})$.

It remains to show that the tag of π_1 is no greater than the tag of γ_{s_1}. Therefore we focus now on the relationship between operation π_1 and garbage-collection operation γ_{s_1}. The propagation phase of π_1 accesses write-quorum $W(\pi_1, s_1)$ of configuration $c(s_1)$, whereas the query phase of γ_{s_1} accesses read-quorum $R(\gamma_{s_1}, s_1)$ of configuration $c(s_1)$. Since $W(\pi_1, s_1) \cap R(\gamma_{s_1}, s_1) \neq \emptyset$, we may fix some $j \in W(\pi_1, s_1) \cap R(\gamma_{s_1}, s_1)$. Let message m_1 from i_1 to j and message m_1' from j to i_1 be as in Lemma 7.4 for the propagation phase of γ_{s_1}.

Let message m_2 from the process running γ_{s_1} to j and message m_2' from j to the process running γ_{s_1} be the messages whose existence is asserted in Lemma 7.5 for the query phase of γ_{s_1}.

We claim that j sends m_1', its message for π_1, before it sends m_2', its message for γ_{s_1}. Suppose for the sake of contradiction that j sends m_2' before it sends m_1'. Assume that $s_t = \mathit{target}(\gamma_{s_1}$. Notice that $s_t > s_1$, since $s_1 \in \mathit{removal\text{-}set}(\gamma_{s_1})$. Lemma 7.5 implies that in any state after j receives m_2, before j sends m_2', $\mathit{cmap}(k)_j \neq \perp$ for all $k \leq s_t$. Since j sends m_2' before it sends m_1', monotonicity of cmap implies that just before j sends m_1', $\mathit{cmap}(k)_j \neq \perp$ for all $k \leq s_t$. Then Lemma 7.4 implies that $\mathit{prop\text{-}cmap}(\pi_1)(\ell) \in C$ for some $\ell \geq s_t$. But this contradicts the choice of s_1, since $s_1 < s_t$. This implies that j sends m_1' before it sends m_2'.

Since j sends m_1' before it sends m_2', Lemma 7.4 implies that, at the time j sends m_2', $\mathit{tag}(\pi_1) \leq \mathit{tag}_j$. Then Lemma 7.5 implies that $\mathit{tag}(\pi_1) \leq \mathit{tag}(\gamma_{s_1})$. From above, we know that $\mathit{tag}(\gamma_{s_1}) \leq \mathit{tag}(\gamma_{s_2-1})$, and $\mathit{tag}(\gamma_{s_2-1}) \leq \mathit{tag}(\pi_2)$, and if π_2 is a write operation then $\mathit{tag}(\gamma_{s_2-1}) < \mathit{tag}(\pi_2)$. Combining the various inequalities then yields both conclusions. □

7.7 Atomicity

Let β be a trace of $\mathcal{S}$ that satisfies the RAMBO environment assumptions, and assume that all read and write operations complete in β. Consider any particular (good) execution α of $\mathcal{S}$ whose trace is β.[3] We define a partial order $\prec$ on read and write operations in β, in terms of the operations' tags in α. Namely, we totally order the writes in order of their tags, and we order each read with respect to all the writes as follows: a read with tag t is ordered after all writes with tags $\leq t$ and before all writes with tags $> t$.

Lemma 7.18 *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 7.17 implies that the tag of the second is greater than the tag of the first. □

Lemma 7.19 *$\prec$ satisfies the four conditions in the definition of atomicity.*

Proof. We begin with Property 2, which as usual in such proofs, is the most interesting thing to show. Suppose for the sake of contradiction that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$. We consider two cases:

[3]The "scope" of these definitions of α and β is just the following two lemmas and their proofs.

1. π_2 is a write operation.
   Since π_1 completes before π_2 starts, Lemma 7.17 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.
2. π_2 is a read operation.
   Since π_1 completes before π_2 starts, Lemma 7.17 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. Property 1 follows from Property 2. Properties 3 and 4 are straightforward. □

Tieing everything together, we conclude with Theorem 7.1.

Theorem 7.1 *Let β be a trace of the system $\mathcal{S}$. If β satisfy the* RAMBO *environment assumptions, then β satisfies the* RAMBO *service guarantees (well-formedness and atomicity).*

Proof. Let β be a trace of $\mathcal{S}$ that satisfies the RAMBO environment assumptions. We argue that β satisfies the RAMBO service guarantees. The proof that β satisfies the RAMBO well-formedness guarantees is straightforward from the code. To show that β satisfies the atomicity condition (as defined in Section 3), assume that all read and write operations complete in β. Let α be an execution of $\mathcal{S}$ whose trace is β. Define the ordering $\prec$ on the read and write operations in β as above, using the chosen α. Then Lemma 7.19 says that $\prec$ satisfies the four conditions in the definition of atomicity. Thus, β satisfies the atomicity condition, as needed. □

8 Implementation of the Reconfiguration Service

In this section, we describe a distributed algorithm that implements the $Recon$ service for a particular object x (and we suppress mention of x). This algorithm is considerably simpler than the $Reader\text{-}Writer$ algorithm. It consists of a $Recon_i$ automaton for each location i, which interacts with a collection of global consensus services $Cons(k, c)$, one for each $k \geq 1$ and each $c \in C$, and with a point-to-point communication service.

$Cons(k, c)$ accepts inputs from members of configuration c, which it assumes to be the $k - 1^{st}$ configuration. These inputs are proposed new configurations. The decision reached by $Cons(k, c)$, which must be one of the proposed configurations, is determined to be the k^{th} configuration.

$Recon_i$ is activated by the joining protocol. It processes reconfiguration requests using the consensus services, and records the new configurations that the consensus services determine. $Recon_i$ also conveys information about new configurations to the members of those configurations, and releases new configurations for use by $Reader\text{-}Writer_i$. It returns acknowledgments and configuration reports to its client.

We first describe the consensus service $Cons(k, c)$ in Section 8.1, which can be implemented using the Paxos consensus algorithm [36]. We then in Section 8.2 describe the $Recon$ automata that, together with the consensus services, implement the reconfiguration service.

8.1 Consensus services

In this subsection, we specify the behavior we assume for consensus service $Cons(k, c)$, for a fixed $k \geq 1$ and $c \in C$. This behavior can be achieved using the Paxos consensus algorithm [36], as described formally in [**?**]. (In the implementation of the $Recon$ service, V will be instantiated as C.) The external signature of $Cons(k, c)$ is given in Figure 9.

We describe the safety properties of $Cons(k, c)$ in terms of properties of a trace β of actions in the external signature. Namely, we define the client 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.

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

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

- At most one $\mathsf{init}(*)_{k,c,i}$ event occurs in β.

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 β.
  - If a $\mathsf{decide}(*)_{k,c,i}$ event occurs in β, 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 β, then $v = v'$.
- *Validity:* If a $\mathsf{decide}(v)_{k,c,i}$ event occurs in β, then it is preceded by an $\mathsf{init}(v)_{k,c,j}$.

We assume that the $Cons(k, c)$ service is implemented using the Paxos algorithm [36], as described formally in [**?**]. This satisfies the safety guarantees described above, based on the safety assumptions:

Theorem 8.1 *If β is a trace of $Paxos$ that satisfies the safety assumptions of $Cons(k, c)$, then β also satisfies the (well-formedness, agreement, and validity) safety guarantees of $Cons(k, c)$.*

8.2 Recon automata

A $Recon_i$ process is responsible for initiating consensus executions to help determine new configurations, for telling the local $Reader\text{-}Writer_i$ process about a newly-determined configuration, and for disseminating information about newly-determined configurations to the members of those configurations. The signature and state of $Recon_i$ appear in Figures 10, and the transitions in Figure 11.

Signature:

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

State:

$status \in \{idle, active\}$, initially $idle$.	$cons\text{-}data \in (\mathbb{N}^+ \rightarrow (C \times C))$: initially $\perp$ everywhere
$rec\text{-}cmap \in CMap$, initially $rec\text{-}cmap(0) = c_0$ and $rec\text{-}cmap(k) = \perp$ for all $k \neq 0$.	$rec\text{-}status \in \{idle, active\}$, initially $idle$
$did\text{-}init \subseteq \mathbb{N}^+$, initially $\emptyset$	$outcome \in \{ok, nok, \perp\}$, initially $\perp$
$did\text{-}new\text{-}config \subseteq \mathbb{N}^+$, initially $\emptyset$	$reported \subseteq C$, initially $\emptyset$
	$failed$, a Boolean, initially $false$

Figure 10: $Recon_i$: Signature and state

Location i joins the $Recon$ service when a $join(recon)$ input occurs. $Recon_i$ responds with a join-ack. $Recon_i$ includes a state variable $rec\text{-}cmap$, which holds a $CMap$: $rec\text{-}cmap(k) = c$ indicates that i knows that c is the

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

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

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

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

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

Output $\mathsf{report}(c)_i$
Precondition:
  $\neg \mathit{failed}$
  $\mathit{status} = \mathit{active}$
  $c \notin \mathit{reported}$
  $S = \{\ell : \mathit{rec\text{-}cmap}(\ell) \in C\}$
  $c = \mathit{rec\text{-}cmap}(\max(S))$
Effect:
  $\mathit{reported} \leftarrow \mathit{reported} \cup \{c\}$

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

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

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

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

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

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

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

Figure 11: Recon_i: Transitions.

kth configuration identifier. If $Recon_i$ has learned that c is the kth configuration identifier, it can convey this to its local $Reader\text{-}Writer_i$ process using a new-config$(c, k)_i$ output action, and it can inform any other $Recon_j$ process, $j \in members(c)$, by sending a $\langle config, c, k \rangle$ message. $Recon_i$ learns about new configurations either by receiving a decide input from a $Cons$ service, or by receiving a *config* or *init* message from another process.

$Recon_i$ receives a reconfiguration request from its environment via a recon$(c, c')_i$ event. Upon receiving such a request, $Recon_i$ determines whether (a) i is a member of the known configuration c with the largest index $k - 1$ and (b) it has not already prepared data for a consensus for the next larger index k. If both (a) and (b) hold, $Recon_i$ prepares such data, consisting of the pair $\langle c, c' \rangle$, where c is the $k - 1$st configuration identifier and c' is the proposed configuration identifier. Otherwise, $Recon_i$ responds negatively to the new reconfiguration request.

$Recon_i$ initiates participation in a $Cons(k, c)$ algorithm when its consensus data are prepared. After initiating participation in a consensus algorithm, it sends *init* messages to inform the other members of c about its initiation of consensus. The other members use this information to prepare to participate in the same consensus algorithm (and also to update their *rec-cmap* if necessary). 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 decide$(c')_{k,i}$ directly from $Cons(k, c)$, it records configuration c' in *rec-cmap* It also determines if 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.

Each consensus service $Cons(k, c)$ is responsible for conveying consensus decisions to $members(c)$. The $Recon_i$ components are responsible for telling $members(c')$ about c' by sending *new-config* messages.

Theorem 8.2 *The Recon implementation guarantees well-formedness, agreement, and validity.*

9 Latency and Fault-Tolerance Assumptions

In this and the following two Sections 10 and 11, 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. We present the results in two groups: Section 10 contains results for executions in which "normal timing behavior" is observed throughout the execution, and Section 11 contains results for executions that "stabilize" so that "normal timing behavior" is observed from some point onward. In this section, we present a series of timing-related assumptions used in the context of both sections.

We formulate these results for the full RAMBO system S' consisting of $Reader\text{-}Writer_i$ and $Joiner_i$ for all i, $Recon_{impl}$ (which consists of $Recon_i$ for all i and $Cons(k, c)$ 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 [39] by allowing arbitrary amounts of time to pass in any state, without changing the state.

9.1 Restricting nondeterminism

RAMBO in its full generality is a highly nondeterministic algorithm. For example, it allows sending of gossip messages at arbitrary times. For the remainder of this paper, 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*. We assume a restricted version of RAMBO in which each $Reader\text{-}Writer_i$, $Joiner_i$, 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 may run at different rates. Moreover, the following conditions hold in all admissible timed executions (those timed executions in which the limit time is ∞):

- *Periodic gossip:* Each $Joiner_i$ whose $status = joining$ 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 $world$ set every time d, according to its local clock.
- *Important Joiner messages:* Each $Joiner_i$ sends a $join$ message immediately to location j, without any time passing on its local clock, when a $\mathsf{join}(rambo, J)_i$ event occurs, 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 $\mathsf{recv}(join)_{j,i}$ event occurs, if $status_i = active$.
  - Just after a $\mathsf{recv}(*, *, *, *, pns, *)_{j,i}$ event occurs, if $pns > pnum2(j)_i$ and $status_i = active$.
  - Just after a $\mathsf{new\text{-}config}(c, k)_i$ event occurs if $status_i = active$ and $j \in world_i$.
  - Just after a read_i, 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 $op.cmap_i$.
  - Just after a $\mathsf{gc}(k)_i$ event occurs, if $j \in members(cmap(k)_i)$.
  - Just after a $\mathsf{gc\text{-}query\text{-}fix}(k)_i$ event occurs, if $j \in members(cmap(k)_i)$.
- *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 $(config,\ c,\ k)$, and a $\mathsf{decide}(c)_{k,*,i}$ event has just occurred, for $j \in members(c) - \{i\}$.
  - The message is of the form $(init,\ c,\ c',\ k)$, and an $\mathsf{init}(c')_{k,c,i}$ event has just occurred, for $j \in members(c) - \{i\}$.
- *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.

We also assume that every garbage-collection operation removes the maximal number of obsolete configurations:

- If a $\mathsf{gc}(k)_i$ event occurs, then immediately prior to the event, there is no $k' > k$ such that $\mathsf{gc}(k')_i$ is enabled.

An alternative to listing these properties is to add appropriate bookkeeping to the various RAMBO automata to ensure these properties. We avoid this approach for the sake of simplicity.

9.2 Normal timing behavior

We also restrict the timing anomalies observed in an execution, specifically, the reliability of the clocks and the latency of message delivery. We define executions that satisfy some "normal" timing behavior from some point onwards. An execution that always satisfies normal timing behavior is viewed as a special case.

Let α be an admissible timed execution, and α' a finite prefix of α. Arbitrary timing behavior is allowed in α': messages may be lost or delivered late, clocks may run at arbitrary rates, and in general any asynchronous behavior may occur. However we assume that after α', good behavior resumes.

Definition 9.1 We say that α is an α'-*normal* execution if the following assumptions hold:

1. *Initial time:* The $\mathsf{join\text{-}ack}_{i_0}$ event occurs at time 0, completing the join protocol for node i_0, the node that created the data object.

2. *Regular timing:* The local clocks of all RAMBO automata (i.e., $Reader\text{-}Writer_i$, $Recon_i$, $Joiner_i$) at all nodes progress at exactly the rate of real time, after α'. Recall from Section 9.1 that the timing of gossip messages and the performance of other locally-controlled events rely on the local clocks. Thus, this single assumptions implies that the timing of all locally-controlled events observes real-time constraints after α'.

3. *Reliable message delivery:* No message sent in α after α' is lost.

4. *Message delay bound:* If a message is sent at time t in α and it is delivered, then it is delivered by time $\max(t, \ell time(\alpha')) + d$.

9.3 Hypotheses for latency results

We now proceed to list various hypotheses that we need for our latency bound results. Notice that none of the assumptions depend on time in α', i.e., during the portion of the execution in which time cannot be reliably measured.

Configuration-Viability. The first hypothesis we define, *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. Similarly 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 takes into account which configurations might still be in use.

If α is a timed execution, we say that configuration c is *installed* in α when for some $k \geq 0$ either of the following holds: (i) $c = c_0$ or (ii) for every $i \in members(c(k-1))$, either a $\mathsf{decide}(c)_{k,c(k-1),i}$ event or a fail_i event occurs in α. That is, configuration c is installed when every non-failed member of some configuration $c(k-1)$ performs a $\mathsf{decide}(c(k))$ event. We now define what it means for an execution to be *(α', e, τ)-configuration-viable*:

Definition 9.2 Let α be an admissible timed execution, and let α' be a finite prefix of α. Let $e, \tau \in \mathbb{R}^{\geq 0}$. Then α is *(α', e, τ)-configuration-viable* if the following holds:
For all i, c, k such that $cmap(k)_i = c$ in some state in α, 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 α.

2. There exists a finite prefix $\alpha_{install}$ of α such that (a) for all $\ell \leq k+1$, configuration $c(\ell)$ is installed in $\alpha_{install}$, (b) no process in $R \cup W$ fails in α at or before time $\ell time(\alpha') + e + 2d + \tau$, (c) no process in $R \cup W$ fails in α at or before time $\ell time(\alpha_{install}) + \tau$.

We say simply that α satisfies τ-configuration-viability if α satisfies $(\emptyset, 0, \tau)$-configuration viability, i.e., configuration-viability holds from the beginning of the execution.

We believe that the configuration-viability assumption is reasonable for a reconfigurable algorithms such as RAMBO, as the system can be reconfigured when quorums seem to be in danger of failing. New configurations should be chosen to minimize the likelihood of failures.

Recon-Spacing. The next property says recon events do not occur too frequently: a $\mathsf{recon}(c,*)$ is only initiated if some quorum of configuration c has received $\mathsf{report}(c)$ events, and after α', each node waits sufficiently long between initiating recon events:

Definition 9.3 Let α be an α'*-normal* execution, and $e \in \mathbb{R}^{\geq 0}$. We say that α satisfies *(α',e)-recon-spacing* if

1. *recon-spacing-1*: For any $\mathsf{recon}(c,*)_i$ event in α there exists a write-quorum $W \in \mathit{write\text{-}quorums}(c)$ such that for all $j \in W$, $\mathsf{report}(c)_j$ precedes the $\mathsf{recon}(c,*)_i$ event in α.[4]
2. *recon-spacing-2*: For any $\mathsf{recon}(c,*)_i$ event in α after α' the preceding $\mathsf{report(c)_i}$ event occurs at least time e earlier.

We say simply that α satisfies e-recon-spacing if it satisfies $(\emptyset, e)$-recon-spacing.

[4]Notice that this property does not depend on a node's local clock; it can be verified simply by collecting gossip from other nodes for which a $\mathsf{report}(c)$ event has occurred.

Join-Connectivity. The hypothesis of *join-connectivity* states a bound on the time for two participants that join the system to learn about each other.

Definition 9.4 Let α be an α'*-normal* execution, $e \in \mathbb{R}^{\geq 0}$. We say that α satisfies *(α',e)-join-connectivity* provided that: for any time t and nodes i, j such that a $\mathsf{join\text{-}ack}(rambo)_i$ and a $\mathsf{join\text{-}ack}(rambo)_j$ occur no later than time t, if neither i nor j fails at or before $\max(t, \ell time(\alpha')) + e$, then by time $\max(t, \ell time(\alpha')) + e$, $i \in world_j$.

We say that α satisfies e-join-connectivity when it satisfies $(\emptyset, e)$-join-connectivity.

We do not think of join-connectivity as a primitive assumption. Rather, it is a property one might expect to show is satisfied by a good join protocol, under certain more fundamental assumptions, for example, sufficient spacing between join requests. We leave it as an open problem to develop and carefully analyze a more involved join protocol.

Recon-Readiness. The next hypothesis says that when a configuration c is proposed by some client, every member of configuration c must have already joined the system at least some time ago.

Definition 9.5 An α'*-normal* execution α satisfies *(α', e)-recon-readiness* if the following property holds: if a $\mathsf{recon}(*, c)_*$ event occurs at time t, then for every $j \in members(c)$:

- A $\mathsf{join\text{-}ack}_j$ event occurs prior to the recon event.
- If the recon occurs after α', then a $\mathsf{join\text{-}ack}_j$ event occurs no later than time $t - (e + 3d)$.

We say simply that α satisfies e-recon-readiness if it satisfies $(\emptyset, e)$-recon-readiness.

GC-Readiness. The last hypothesis ensures that after the system stabilizes, a node initiates a read, write, or garbage-collection operation only if it has joined sufficiently long ago.

Definition 9.6 We say that an α'*-normal* execution α satisfies *(α', e, d)-gc-readiness* if the following property holds: if for some i a gc_i event occurs in α after α' at time t, then a $\mathsf{join\text{-}ack}_j$ event occurs no later than time $t - (e + 3d)$.

Notice that the gc action is an internally-controlled action, and hence in this case, the hypothesis could be enforced via explicit reference to the local clock.

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

Definition 9.7 Let α be an admissible timed execution. We say that α satisfies *infinite reconfiguration* provided for every $k \in \mathbb{N}^+$, α contains a $\mathsf{decide}(*)_{k,*,*}$ event.

10 Latency and Fault-Tolerance: Normal Timing Behavior

In this section, we present conditional performance results for the case where normal timing behavior is satisfied throughout the execution. The main result of this section, Theorem 10.7, shows that every read and write operation completes within $8d$ time, despite concurrent failures and reconfigurations.

For this entire section, we fix α to be an α'*-normal* admissible timed execution where α' is an empty execution. That is, α exhibits normal timing behavior throughout the execution. We fix $e \in \mathbb{R}^{\geq 0}$. Also, for a timed execution α, we let $time(\pi)$ stand for the real time at which the event π occurs in α.

10.1 Joining

This first lemma says that a process receiving a report must be "old enough", that is, they have joined at least time e earlier.

Lemma 10.1 *Assume that α satisfies e-recon-readiness, $c \in C$, $c \neq c_0$, $i \in I$. Suppose that a* $\mathsf{report}(c)_i$ *event occurs at time t in α. Then a* $\mathsf{join\text{-}ack}(rambo)_i$ *event occurs by time $t - e$.*

Proof. Assume that α, c, and i are as given, and that $\mathit{rec\text{-}cmap}(k)_i = c$ 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 $i \in \mathit{members}(c)$, then the conclusion follows immediately from *e-recon-readiness*. If $i \in \mathit{members}(c')$ and $c' \neq c_0$, then again the conclusion follows from *e-recon-readiness* and the fact that c' was proposed no later than time t.

The only other possibility is that $i \in \mathit{members}(c')$ and $c' = c_0$. Then a $\mathsf{join\text{-}ack}_i$ event occurs at time 0, prior to any $\mathsf{join\text{-}ack}_*$ event by any member of configuration c. But *e-recon-readiness* implies that every member of configuration c performs a join-ack no later than time $t - e$, as needed. ☐

10.2 Propagation of information.

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 do not fail, then any information that i has about configurations is conveyed to j within time $2d$.

Lemma 10.2 *Assume that α satisfies e-join-connectivity, $t \in \mathbb{R}^{\geq 0}$, $t \geq e$. Suppose:*

1. $\mathsf{join\text{-}ack}(rambo)_i$ *and* $\mathsf{join\text{-}ack}(rambo)_j$ *both occur in α 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, $\mathit{cmap}(k)_i \neq \bot$, then by time $t + 2d$, $\mathit{cmap}(k)_j \neq \bot$.*
2. *If by time t, $\mathit{cmap}(k)_i = \pm$, then by time $t + 2d$, $\mathit{cmap}(k)_j = \pm$.*

Proof. Follows by join-connectivity and regular gossip. ☐

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 (a) the report event and (b) the time of j's joining.

Theorem 10.3 *Assume that α satisfies 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 α, where $c = \mathit{rec\text{-}cmap}(k)_i$, and i does not fail by* $\max(t, t') + d$*.*
2. $\mathsf{join\text{-}ack}(rambo)_j$ *occurs in α by time $t' - e$, and j does not fail by time* $\max(t, t') + 2d$*.*

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

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

Lemma 10.1 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 $\max(t, t')$, $\mathit{cmap}(k)_i \neq \bot$. This is because the time of the $\mathit{report}(c)_i$ is $\leq \max(t, t')$, when the $\mathsf{report}(c)_i$ occurs, $\mathit{rec\text{-}cmap}(k)_i \neq \bot$, and within 0 time, this information gets conveyed to $\mathit{Reader\text{-}Writer}_i$.

Therefore, we may apply Lemma 10.2, with the t in that theorem instantiated to $\max(t, t')$, to conclude that by time $\max(t, t') + 2d$, $\mathit{cmap}(k)_j \neq \bot$. This yields the conclusion. ☐

10.3 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 10.4 *Assume that α satisfies $5d$-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 $\mathsf{report}(c)_i$ event occurs at time t in α, where $c = \mathit{rec\text{-}cmap}(k)_i$.*
2. *c' is configuration $k-1$ in α.*

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

Proof. The behavior of the 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 α. 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 $t + 4d$. □

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 10.5 *Let α be an admissible timed execution 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 α, where $c = \mathit{rec\text{-}cmap}(k)_i$.*
2. *$\mathsf{join\text{-}ack}(\mathit{rambo})_j$ occurs in α by time $t - e$.*

Then:

1. *If $k > 0$ and j does not fail by time $t+2d$, then by time $t+2d$: (a) $\mathit{cmap}(k-1)_j \neq \bot$ and (b) $\mathit{cmap}(\ell)_j = \pm$ 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) $\mathit{cmap}(k)_j \neq \bot$ and (b) $\mathit{cmap}(\ell)_j = \pm$ 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 $\mathit{cmap}(0)_j \neq \bot$ 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), $\mathit{cmap}(0)_j \neq \bot$ 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 10.4 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 $k - 1 \geq 1$, *e-recon-readiness* implies that a $\mathsf{join\text{-}ack}(\mathit{rambo})_{i''}$ event occurs at least time e prior to the $\mathsf{report}(c')_{i'}$ event. Then by inductive hypothesis, Part 2, by time $\mathit{time}(\mathsf{report}(c')_{i'}) + 6d$, $\mathit{cmap}(k-1)_{i''} \neq \bot$ and $\mathit{cmap}(\ell)_{i''} = \pm$ for all $\ell < k-1$. By $6d$*-recon-spacing*, $\mathit{time}(\mathsf{recon}(c', c)_{i'}) \geq \mathit{time}(\mathsf{report}(c')_{i'}) + 6d$, and so $t = \mathit{time}(\mathsf{report}(c)_i) \geq \mathit{time}(\mathsf{report}(c')_{i'}) + 6d$. Therefore, by time t, $\mathit{cmap}(k-1)_{i''} \neq \bot$ and $\mathit{cmap}(\ell)_{i''} = \pm$ for all $\ell < k-1$.

Now we apply Lemma 10.2 to i'' and j, with t in the statement of Lemma 10.2 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 Part 2, that is, that i does not fail by time $t+d$ and j does not fail by time $t+6d$. Theorem 10.3 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 where $k-1$ is in the removal set (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 10.6 *Let α be an admissible timed execution 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 α, 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 $k \geq 1$, then *e-recon-readiness* implies that join-ack$(rambo)_j$ occurs in α by time $t - e$. Then the conclusions follow from Lemma 10.5. □

10.4 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 10.7 *Assume that α satisfies e-recon-readiness, e-join-connectivity, $(12d+\varepsilon)$-recon-spacing, $11d$-configuration-viability, and infinite reconfiguration, $i \in I$, $\epsilon, t \in \mathbb{R}^{+}$. Assume that a* read$_i$ *(resp.,* write$(*)_i$*) event occurs at time t, and* join-ack$_i$ *occurs strictly before time $t-(e+8d)$. Then the corresponding* read-ack$_i$ *(resp.,* write-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 α; by infinite reconfiguration, this sequence exists. For each $k \geq 0$, let π_k be the first recon$(c_k, c_{k+1})_*$ event in α, let i_k be the location at which this occurs, and let ϕ_k be the corresponding, preceding 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 $s_k \in members(c_k)$ such that s_k does not fail by time $10d$ after the time of ϕ_{k+1}. The fact that this is possible follows from $11d$-viability (because the report event ϕ_{k+1} happens at most time d after the final 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 ψ be the read$_i$, write$_i$ or query-fix$_i$ event that begins the phase.

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

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

Next, we claim that by $time(\phi_k) + 6d$, $cmap(k)_{s_k} \neq \bot$ and $cmap(\ell)_{s_k} = \pm$ for all $\ell < k$; this follows from Lemma 10.6, Part 2, applied with $i = i_k$ and $j = s_k$, because i_k does not fail befire π_k, and because s_k does not fail by time $10d$ after ϕ_{k+1}.

Next, we show that in the pre-state of ψ, $cmap(k)_i \neq \bot$ and $cmap(\ell)_i = \pm$ for all $\ell < k$: We apply Lemma 10.2 to s_k and i, with t in that lemma set to $\max(time(\phi_k) + 6d, time(\mathsf{join\text{-}ack}_i) + e)$. This yields that, by time $\max(time(\phi_k) + 6d, time(\mathsf{join\text{-}ack}_i) + e) + 2d$, $cmap(k)_i \neq \bot$ and $cmap(\ell)_i = \pm$ for all $\ell < k$. Our choice of k implies that $time(\phi_k) + 8d < time(\psi)$. Also, by assumption, $time(\mathsf{join\text{-}ack}_i) + e + 2d < t$. And $t \leq time(\psi)$. So, $time(\mathsf{join\text{-}ack}_i) + e + 2d < time(\psi)$. Putting these inequalities together, we obtain that $\max(time(\phi_k) + 6d, time(\mathsf{join\text{-}ack}_i) + e) + 2d < time(\psi)$. It follows that, in the pre-state of ψ, $cmap(k)_i \neq \bot$ and $cmap(\ell)_i = \pm$ for all $\ell < k$, as needed.

Now, by choice of k, we know that $time(\psi) \leq time(\phi_{k+1}) + 8d$. The recon-spacing condition implies that $time(\pi_{k+1})$ (the first recon event that requests the creation of the $(k+2)^{nd}$ configuration) is $> time(\phi_{k+1}) + 12d$. Therefore, for an interval of time of length $> 4d$ after ψ, 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 ψ completes with at most one additional delay (of $2d$) 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 ϕ_{k+1}. Then at most $8d$ time elapses from ϕ_{k+1} until ψ. 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 $11d$. □

11 Latency and Fault-Tolerance: Eventually-Normal Timing Behavior

In this section, we present conditional performance results for the case where eventually the network stabilizes and normal timing behavior is satisfied from some point on. The main result of this section, Theorem 11.20, is analogous to Theorem 10.7 in that it shows that every read and write operation completes within $8d$ time, despite concurrent failures and reconfigurations. Unlike Theorem 10.7, it shows that this good performance is achieved despite the fact that initially, good timing behavior does not hold. Good performance is achieved by every read and write operation that begins sufficiently long after normal timing behavior resumes.

For this entire section, we fix α to be an α'-*normal* executions. That is, α exhibits normal timing behavior after α'. We fix $e \in \mathbb{R}^{\geq 0}$. We assume throughout this section that execution α satisfies the following hypotheses:

- $(\alpha',\ e)$-*join-connectivity*,
- $(\alpha',\ e)$-*recon-readiness*,
- $(\alpha',\ e)$-*gc-readiness*,
- $(\alpha',\ 13d)$-*recon-spacing*,
- $(\alpha',\ e,\ 22d)$-*configuration-viability*,
- *infinite-reconfiguration*.

As a point of notation, throughout this section we let $T_{GST} = \ell time(\alpha') + e + 2d$. That is, T_{GST} represents the time $(e + 2d)$ after the system has stabilized.

Also, when we refer s as the state *after* time t, we mean that s is the last state in a prefix of α that includes every event that occurs at or prior to time t, and no events that occur after time t. When we refer to s as the state *at* time t, we mean that s is the last state in some prefix of α that ends at time t; it may include any subset of events that occur exactly at time t.

11.1 Propagation of information.

We begin by introducing the notion of information being in the "mainstream." When every non-failed node that has joined sufficiently long ago learns about some CMap cm, we say that cm is in the mainstream:

Definition 11.1 Let cm be a CMap, and $t \geq 0$ a time. We say that cm is *mainstream* after t if the following condition holds:
For every i such that (1) a join-ack$_i$ occurs no later than $t - e - 2d$, and (2) i does not fail until after time t: $cm \leq cmap_i$ after time t.

This concept will be useful in a variety of contexts throughout this section. For example, it allows us to determine precisely what is known at time $\max(t, T_{GST})$ to the members of some configuration c initially proposed at time t:

Lemma 11.2 *Assume that cm is mainstream after some time $t \geq 0$. If c is a configuration that was initially proposed no later than time t, then for every non-failed $i \in members(c)$, $cm \leq cmap_i$ after time* $\max(t, T_{GST})$.

Proof. Fix some $i \in members(c)$. By recon-readiness, we know that a join-ack$_i$ occurs no later than time $t - (e + 3d)$ if $t > \ell time(\alpha')$, and no later than time $\ell time(\alpha')$, otherwise. Thus the claim follows from the definition of "mainstream" with respect to time $\max(t, T_{GST})$. □

Similarly, we conclude that if i is a member of some configuration c, then we can specify certain conditions under which $cmap_i$ becomes mainstream:

Lemma 11.3 *Let c be a configuration that is initially proposed no later than time t, and assume that $i \in members(c)$. If i does not fail until after time $\max(t, T_{GST}) + d$ and $cm = cmap_i$ at time $\max(t, T_{GST})$, then cm is mainstream after* $\max(t, T_{GST}) + 2d$.

Proof. By recon-readiness, we know that i performs a join-ack$_i$ no later than time $\max(t - (e + 3d), T_{GST} - (e + 2d))$. In order to show that cm is mainstream, we need to show that $cm \leq cmap_j$ for every j that performs a join-ack$_j$ no later than time $\max(t, T_{GST}) - e$ and that does not fail by time $\max(t, T_{GST}) + 2d$. Fix some such j. By join-connectivity, we know that j is in $world_i$ by time $\max(t, T_{GST})$. From this we conclude that j receives a message from i by time $\max(t, T_{GST}) + 2d$, resulting in the desired outcome. □

The main result in this subsection is Theorem 11.6, which shows that if some CMap cm is mainstream at some time t_1, then at all times $t_2 \geq t_1 + 2d$, CMap cm remains mainstream. We focus on times that occur sufficiently after T_{GST}; prior to T_{GST}, there may be no propagation of information.

Definition 11.4 We say that a recon$(*, c)$ event is *successful* if at some time afterwards a decide$(c)_{k,i}$ event occurs for some k and i.

We first consider a special case of Theorem 11.6 where a successful recon event occurs at time t_2.

Lemma 11.5 *Fix times $t_2 \geq t_1 \geq T_{GST} + 2d$. Assume that some CMap cm is mainstream after t_1 and that a successful* recon$_*$ *event occurs at time t_2. Then cm is mainstream after $t_2 + 2d$.*

Proof. We prove the result by induction on the number of successful recon events that occur at or after time t_1.

We consider both the base case and the inductive step simultaneously (with differences in the base case in parentheses). Consider the $(n + 1)^{\text{st}}$ successful recon event in α that occurs at or after time t_1. (For the base case, $n = 0$.) Assume this event occurs at time t_{rec}; fix h as the old configuration and h' as the new configuration.

Inductively assume the following: if event π is one of the first n successful recon events in α that occur at some time $t_{pre} \geq t_1$, then cm is mainstream after $t_{pre} + 2d$.

We need to show that cm is mainstream after $t_{rec} + 2d$. That is, we need to show that after time $t_{rec} + 2d$, $cm \leq cmap_i$ for every i such that (1) a join-ack$_i$ occurs by time $t_{rec} - e$, and (2) i does not fail by time $t_{rec} + 2d$. Fix some such i.

If $n > 0$, let t_{pre} be the time of the n^{th} successful recon$(*, h)$ event. (In the base case, let $t_{pre} = t_1$.) If $n > 0$, the inductive hypothesis shows that cm is mainstream after $t_{pre} + 2d$. (In the base case, by assumption cm is mainstream after t_{pre}.)

Choose some node $j \in members(h)$ such that j does not fail at or before $t_{rec} + 2d$; configuration-viability ensures that such a node exists. Since h is the old configuration, we can conclude that it was initially proposed no later than time $t_{pre} \leq t_{pre} + 2d$, and thus Lemma 11.2 implies that $cm \leq cmap_j$ after time $t_{pre} + 2d$. (In the base case, it is easy to see that configuration h was proposed no later than time t_{pre}, as we are considering the first recon event after time $t_1 = t_{pre}$, and hence it follows that $cm \leq cmap_j$ after time t_{pre}.) Recon-spacing ensures that $t_{pre} + 2d \leq t_{rec}$, and hence $cm \leq cmap_j$ after time t_{rec}.

Finally, recon-readiness guarantees that a join-ack$_j$ occurs no later than time $t_{pre} - (e + 2d)$, and hence by join-connectivity, we conclude that $i \in world_j$ by time t_{rec}, and hence sometime in the interval $(t_{rec}, t_{rec} + d]$, j sends a gossip message to i, ensuring that i receives cm no later than time $t_{rec} + 2d$. □

We now generalize this result to all times t_2:

Theorem 11.6 *Assume that t_1 and t_2 are times such that:*

- $t_1 \geq T_{GST} + 2d$;
- $t_2 \geq T_{GST} + 6d$; *and*
- $t_2 \geq t_1 + 2d$.

If CMap cm is mainstream after t_1, then cm is mainstream after t_2.

Proof. Choose configuration c to be the configuration with the largest index such that a successful recon$(*, c)$ event occurs at or before time $t_2 - 4d$. If no such configuration exists, let $c = c_0$. Assume that this successful recon$(*, c)$ event occurs at time t_{rec}. Note that by the choice of c, no successful recon$(c, *)$ event occur at or before time $t_2 - 4d$. We now show that for every non-failed $i \in members(c)$, $cm \leq cmap_i$ after $t_2 - 2d$. Fix some such i. There are three cases to consider.

1. $c = c_0$:
   Recall that i_0 is the only member of c_0, and performs a join-ack$_{i_0}$ at time 0. Since cm is mainstream after t_1, and we have assumed that $i = i_0$ does not fail until after $t_1 \geq e + 2d$, then $cm \leq cmap_{i_0}$ after time t_1, and hence also after time $t_2 - 2d$.
2. The successful recon$(*, c)$ event occurs after time t_1:
   Since $t_{rec} > t_1$, Lemma 11.5 shows that cm is mainstream after $t_{rec} + 2d$. Since c was initially proposed at time $t_{rec} < t_{rec} + 2d$, Lemma 11.2 implies that for every non-failed member i of configuration c, $cm \leq cmap_i$ after time $t_{rec} + 2d$, and hence after time $t_2 - 2d \geq t_{rec} + 2d$.
3. The successful recon$(*, c)$ event occurs at or before time t_1:
   Since cm is mainstream after t_1, and since configuration c was proposed no later than time t_1, we can conclude by Lemma 11.2 that for every non-failed $i \in members(c)$, $cm \leq cmap_i$ after time t_1, and hence after $t_2 - 2d$.

Configuration-viability guarantees that some member of configuration c does not fail until at least $4d$ after the next configuration is installed. Since no successful recon$(c, *)$ event occurs at or before time $t_2 - 4d$, we can conclude that some node, $j \in members(c)$ does not fail at or before time t_2.

Since configuration c is proposed no later than time $t_2 - 4d$, and since j does not fail until after time t_2, we can conclude from Lemma 11.3 that $cmap_i$ after time $t_2 - 2d$ is mainstream after time t_2. Since $cm \leq cmap_i$ at time $t_2 - 2d$, the result follows. □

11.2 Configuration viability.

In this subsection, we first define an event $\mathsf{gc\text{-}ready}(k)$, for every $k > 0$, that indicates when a $\mathsf{gc}(k)_*$ event may occur. (Garbage-collection operations may, however, occur prior to this event.) We then show in Theorem 11.11 that for every $k \geq 0$, configuration $c(k)$ remains viable for at least $16d$ time after the $\mathsf{gc\text{-}ready}(k+1)$ event.

Definition 11.7 *Define the* $\mathsf{gc\text{-}ready}(k)$ *event for* $k > 0$ *to be the first event in* α *after which,* $\forall \ell \leq k$*, the following hold: (i) configuration* $c(\ell)$ *is installed, and (ii) for all non-failed* $i \in \mathit{members}(c(k-1))$*,* $\mathit{cmap}(\ell)_i \neq \bot$*.*

The first lemma shows that soon after a configuration is installed, every node that joined sufficiently long ago learns about the new configuration.

Lemma 11.8 *Assume that configuration* $c(k)$ *is installed at time* $t \geq 0$*. Then there exists a CMap* cm *such that* $cm(k) \neq \bot$ *and* cm *is mainstream after* $\max(t, T_{GST}) + 2d$*.*

Proof. Configuration-viability guarantees that there exists a read-quorum $R \in \mathit{read\text{-}quorums}(c(k-1))$ such that no node in R fails at or before time $\max(t, T_{GST}) + d$. Choose some node $j \in R$.

Since configuration $c(k)$ is installed at time t, we can conclude that after time t, and hence also after time $\max(t, T_{GST})$, $\mathit{cmap}(k)_j \neq \bot$. Since configuration $c(k-1)$ is initially proposed no later than time $\max(t, T_{GST})$, we conclude by Lemma 11.3 that $\mathit{cmap}(k)_j$ is mainstream after time $\max(t, T_{GST}) + 2d$. $\Box$

The next lemma shows that for configuration with index k, soon after all configurations smaller than k are installed, a $\mathsf{gc\text{-}ready}(k)$ event occurs.

Lemma 11.9 *Let* c *be a configuration with index* k*, and assume that for all* $\ell \leq k$*, configuration* $c(\ell)$ *is installed in* α *by time* t*. Then* $\mathsf{gc\text{-}ready}(k)$ *occurs by time* $\max(t, T_{GST}) + 6d$*.*

Proof. Recall that $\mathsf{gc\text{-}ready}(k)$ is the first event after which (i) all configurations with index $\leq k$ have been installed, and (ii) for all $\ell < k$, for all non-failed members of configuration $c(k-1)$, $\mathit{cmap}(\ell) \neq \bot$. The first part occurs by time t by assumption. We need to show that the second part holds by time $\max(t, T_{GST}) + 6d$.

For every configuration $c(\ell)$ with index $\ell \leq k$, let t_ℓ be the time at which configuration $c(\ell)$ is installed; by assumption $\max(t_i) \leq t$.

For each $\ell \leq k$, we can conclude by Lemma 11.8 that there is some cm where $cm(\ell) \neq \bot$ that is mainstream after $\max(t_\ell, T_{GST}) + 2d$. We conclude from Theorem 11.6 that cm_ℓ is still mainstream after $\max(t, T_{GST}) + 6d$.

Since configuration $c(k-1)$ was proposed and installed prior to time $\max(t, T_{GST}) + 6d$, we conclude by Lemma 11.2 that for every non-failed $j \in \mathit{members}(c(k-1))$, for every $\ell \leq k$, $cm_\ell \leq \mathit{cmap}_j$ after time $\max(t, T_{GST}) + 6d$, as required. $\Box$

As a corollary, we notice that if no $\mathsf{gc\text{-}ready}(k+1)$ occurs in α, then configuration $c(k)$ is always viable.

Corollary 11.10 *For some* $k \geq 0$*, assume that no* $\mathsf{gc\text{-}ready}(k+1)$ *event occurs in* α*. Then there exists a read-quorum* R *and a write-quorum* W *of configuration* $c(k)$ *such that no node in* $R \cup W$ *fails in* α*.*

Proof. Assume that for some $\ell \leq k+1$, configuration $c(\ell)$ is not installed in α. Then the claim follows immediately from configuration viability. Assume, instead, that for every $\ell \leq k+1$, configuration $c(\ell)$ is installed in α. Then by Lemma 11.9, an $\mathsf{gc\text{-}ready}(k+1)$ event occurs in α, contradicting the hypothesis. $\Box$

Finally, we show that if a $\mathsf{gc\text{-}ready}(k+1)$ event does occur, then configuration $c(k)$ remains viable until at least $16d$ after the $\mathsf{gc\text{-}ready}(k+1)$ event.

Theorem 11.11 *For some* $k \geq 0$*, assume that* $\mathsf{gc\text{-}ready}(k+1)$ *occurs at time* t*. Then there exists a read-quorum* R *and a write-quorum* W *of configuration* $c(k)$ *such that no node in* $R \cup W$ *fails by time* $\max(t, T_{GST}) + 16d$*.*

Proof. Let t' be the minimal time such that every configuration with index $\leq k+1$ is installed no later than time t'. We conclude from Lemma 11.9 that the gc-ready$(k+1)$ event occurs by time $\max(t', T_{GST}) + 6d$; that is, $t \leq \max(t', T_{GST}) + 6d$.

Configuration-viability guarantees that there exists a read-quorum R and a write-quorum W of configuration $c(k)$ such that either: Case (1): no process in $R \cup W$ fails in α, or Case (2): there exists a finite prefix, $\alpha_{install}$ of α such that for all $\ell \leq k+1$, configuration $c(\ell)$ is installed in $\alpha_{install}$ and no process in $R \cup W$ fails in α by: (a) $\ell time(\alpha_{install}) + 22d$, or (b) $T_{GST} + 22d$. In Case 1, we are done.

We now consider the second case. Since t' is the minimal time such that every configuration $\leq k+1$ is installed by time t', we can conclude that $t' \leq \ell time(\alpha_{install})$, from which the claim follows immediately. □

11.3 Garbage collection.

In this subsection, we analyze the performance of garbage-collection operations. The main result of this section, Theorem 11.17, shows that every garbage-collection operation completes within $4d$ time. The proof is structured as an induction argument: if every earlier garbage collection has completed within $4d$ time, then the next garbage collection completes within $4d$ time. More specifically:

Definition 11.12 For time $t \geq 0$, we say that execution α satisfies the *gc-completes* hypothesis at time t if every gc event ρ that satisfies the following conditions completes no later than $\max(time(\rho), T_{GST}) + 4d$:

- $time(\rho) < t$.
- Event ρ is performed by some node that does not fail prior to time $\max(time(\rho), T_{GST}) + 4d$.

We first show that under certain circumstances, a garbage-collection operation begins. We consider some prefix of execution α that ends in some event ρ, and show that if the state after ρ satisfies certain conditions (e.g., there is no ongoing garbage collection, and there is garbage to collect), then a garbage-collection begins immediately.

Lemma 11.13 *For some node i, for times $t_1, t_2 \in \mathbb{R}^{\geq 0}$, for some $k > 0$, for some event ρ that occurs at time t_2, assume that:*

1. *A* gc-ready(k) *event occurs at time t_1.*
2. *Event ρ occurs after the* gc-ready(k) *event.*
3. *i does not fail at or before time t_2.*
4. *i is a member of configuration $c(k-1)$.*
5. *(No garbage collection is ongoing:) Immediately after event ρ, $gc.phase_i =$* idle.
6. *(There is garbage to collect:) After time t_2, $cmap(k-1)_i \neq \pm$.*

Then for some $k' \geq k$, i performs a gc$(k')_i$ *at time t_2.*

Proof. Assume for the sake of contradiction that no gc$(*)_i$ event occurs at time t_2 after event ρ. We examine in turn the preconditions for gc$(k)_i$ immediately after all the events that occur at time t_2. Let s be the state of the system after time t_2.

1. $\neg s.failed_i$: By Assumption 3 on i.
2. $s.status_i = active$: Node i is a member of configuration $c(k-1)$ (Assumption 4), which is proposed and installed no later than time t_1 when the gc-ready(k) event occurs. Hence, by recon-readiness we conclude that a join-ack$_i$ occurs no later than time $t_1 - (e + 3d)$, and hence prior to t_2. This also satisfies the gc-readiness hypothesis.
3. $s.gc.phase_i = idle$: By Assumption 5 no garbage collection is ongoing immediately after ρ; by assumption no garbage collection is initiated by i at time t_2 after the event ρ.
4. $\forall \ell < k : s.cmap(\ell)_i \neq \bot$: By the definition of gc-ready(k), Part (ii), we know that for all $\ell \leq k$, for all non-failed $j \in members(c(k-1))$, $cmap(\ell)_j \neq \bot$ immediately after the gc-ready(k) event. Node i satisfies both requirements, and later CMap updates do not change this fact.

5. $s.cmap(k-1)_i \in C$: We have already shown (Part 4) that $s.cmap(k-1)_i \neq \bot$. By Assumption 6, $s.cmap(k-1)_i \neq \pm$.
6. $s.cmap(k)_i \in C$: We have already shown (Part 4) that $s.cmap(k)_i \neq \bot$. Since $s.cmap_i \in \mathit{Usable}$ (Invariant 2), and since $s.cmap(k-1)_i \neq \pm$ (Assumption 6), we can conclude (Lemma 2.1) that $s.cmap(k)_i \neq \pm$.

Since enabled events occur in zero time (by assumption), we conclude that a $\mathsf{gc}(k')_i$ event occurs at time t_2, contradicting our assumption that no such event occurs. Moreover, by the restrictions on non-determinism (Section 9.1), we conclude that $k' \geq k$. □

Next, we show that if we assume the gc-completes hypothesis, then within $8d$ after a $\mathsf{gc\text{-}ready}(k+1)$ event, some node in configuration $c(k)$ has already removed configuration $c(k)$. Essentially, this lemma relies on Lemma 11.13 to show that some garbage collection is started, and the gc-completes hypothesis to show that the garbage collection completes.

Lemma 11.14 *For some time $t_2 \geq 0$, assume that α satisfies the* gc-completes *hypothesis for time t_2. Assume that for some $k \geq 0$, a* $\mathsf{gc\text{-}ready}(k+1)$ *event occurs at time t_1 such that* $\max(t_1, T_{GST}) + 4d < t_2$.

Then for some node $i \in members(c(k))$ that does not fail at or before time $\max(t_1, T_{GST}) + 10d$*: we conclude that $cmap(k)_i = \pm$ after time* $\max(t_1, T_{GST}) + 8d$.

Proof. We know from Theorem 11.11 that there exists a read-quorum R and a write-quorum W of configuration $c(k)$ such that no node in $R \cup W$ fails at or before time $\max(t_1, T_{GST}) + 16d$. Choose $i \in R \cup W$.

Assume for the sake of contradiction that $cmap(k)_i \neq \pm$ after time $\max(t_1, T_{GST}) + 8d$. We argue that i begins a garbage-collection operation no later than $\max(t_1, T_{GST}) + 4d$; the contradiction—and conclusion—then follow from the gc-completes hypothesis. There are two cases depending on whether $gc.phase_i = idle$ or $active$ immediately after the $\mathsf{gc\text{-}ready}(k+1)$ event:

- *Case 1:* Assume that $gc.phase_i = idle$ immediately after the $\mathsf{gc\text{-}ready}(k+1)$ event:
  Notice that all the conditions of Lemma 11.13 are satisfied: (1) a $\mathsf{gc\text{-}ready}(k+1)$ event occurs at time t_1; (2) let ρ be the $\mathsf{gc\text{-}ready}(k+1)$ event; (3) i does not fail at or before time t_1; (4) i is a member of configuration $c(k)$; (5) there is no ongoing garbage collection at i (by Case 1 assumption); (6) and $cmap(k)_i \neq \pm$, since we assumed for the sake of contradiction that $cmap(k)_i \neq \pm$ at some time $> t_1$ and CMap updates do not invalidate this fact. Thus we conclude that i performs a $\mathsf{gc}(k')_i$ event for some $k' > k$ at time t_1.

- *Case 2:* Assume that $gc.phase_i = active$ immediately after the $\mathsf{gc\text{-}ready}(k+1)$ event:
  This implies that some event $\rho = \mathsf{gc}(*)_i$ with no matching gc-ack occurs no later than time t_1. By the gc-completes hypothesis, since (1) $time(\rho) \leq t_1 < t_2$; and (2) i does not fail at or before time $\max(t_1, T_{GST}) + 4d$: we conclude that a $\mathsf{gc\text{-}ack}_i$ occurs at some time t_{ack} such that $t_{ack} \leq \max(t_1, T_{GST}) + 4d$.

  At this point, we again invoke Lemma 11.13: (1) a $\mathsf{gc\text{-}ready}(k+1)$ event occurs at time t_1; (2) let ρ be the gc-ack event; (3) i does not fail at or before time t_{ack}; (4) i is a member of configuration $c(k)$; (5) there is no ongoing garbage collection at i; (6) and $cmap(k)_i \neq \pm$, since we assumed for the sake of contradiction that $cmap(k)_i \neq \pm$ at some time $> t_1$ and CMap updates do not invalidate this fact. We conclude that i performs a $\mathsf{gc}(k')_i$ event for some $k' > k$ at time t_{ack}.

In either case, i performs a $\mathsf{gc}(k')$ event for some $k' > k$ no later than time $\max(t_1, T_{GST}) + 4d < t_2$. Moreover, i does not fail at or before $\max(t_1, T_{GST}) + 8d$. We conclude via the gc-completes hypothesis that a $\mathsf{gc\text{-}ack}(k')_i$ event occurs no later than $\max(t_1, T_{GST}) + 8d$, resulting in $cmap(k)_i = \pm$ after $\max(t_1, T_{GST}) + 8d$. □

We now show that, if the gc-completes hypothesis holds, every configuration remains viable for as long as it is being used by any $cmap$. This lemma depends on Lemma 11.14 to show that a configuration with index k is removed soon after the gc-ready$(k+1)$ event, and also Theorem 11.11 to show that configuration $c(k)$ remain viable long enough after the gc-ready$(k+1)$ event; finally, it uses Theorem 11.6 to show that once a configuration is removed, every other node learns about it sufficiently rapidly.

Lemma 11.15 *Fix a time $t \geq T_{GST} + 13d$, and assume that α satisfies the* gc-completes *hypothesis for time t. Assume that for some non-failed node i that performs a* join-ack *no later than time $t - (e + 3d)$, for some $k \geq 0$, $cmap(k)_i \in C$ at time t. Then there exists a read-quorum R and a write-quorum W of configuration $c(k)$ such that no node in $R \cup W$ fails by time $t + 3d$.*

Proof. First, consider the case where no gc-ready$(k+1)$ event occurs in α. Corollary 11.10 implies that there exists a read-quorum R and a write-quorum W of configuration $c(k)$ such that no node in $R \cup W$ fails in α.

Next, consider the case where a gc-ready$(k+1)$ event occurs in α at some time $t_{ready} \geq t - 13d$. Then Theorem 11.11 implies that there exists a read-quorum R and a write-quorum W of configuration $c(k)$ such that no node in $R \cup W$ fails by time $\max(t_{ready}, T_{GST}) + 16d$, implying the desired result.

Finally, consider the case where a gc-ready$(k+1)$ event occurs in α at some time $t_{ready} < t - 13d$. We show that this implies that $cmap(k)_i = \pm$ by time t, resulting in a contradiction. That is, this third case cannot occur.

To begin with, Lemma 11.14 demonstrates that for some $j \in members(c(k))$ that does not fail at or before time $\max(t_{ready}, T_{GST}) + 10d$, $cmap(k)_j = \pm$ after $\max(t_{ready}, T_{GST}) + 8d$. Let cm be j's $cmap$ at this point. Since configuration $c(k)$ is proposed prior to time t_{ready}, Lemma 11.3 indicates that cm is mainstream after $\max(t_{ready}, T_{GST}) + 10d$. And since $t - d \geq \max(t_{ready}, T_{GST}) + 12d$, we conclude by Theorem 11.6 that cm is still mainstream after time $t - d$. Since i performs a join-ack$_i$ no later than time $t - (e + 3d)$ and does not fail prior to time t, we conclude that $cm \leq cmap(k)_i$ after time $t - d$, resulting in a contradiction. □

We can now begin to analyze the actual latency of a garbage-collection operation. We first show that if sufficient configurations remain viable, then the operation completes with $4d$ time.

Lemma 11.16 *Let $t \geq 0$ be a time. Let i be a node that does not fail at or before time $\max(t, T_{GST}) + 4d$. Assume that i initiates garbage-collection operation γ at time t with a* gc$(k)_i$ *event.*

Additionally, assume that for every $\ell \in removal\text{-}set(\gamma) \cup \{k\}$, there exists a read-quorum R_ℓ and a write-quorum W_ℓ of configuration $c(\ell)$ such that no node in $R_\ell \cup W_\ell$ fails by time $\max(t, T_{GST}) + 3d$.

Then a gc-ack$(k)_i$ *event occurs no later than $\max(t, T_{GST}) + 4d$.*

Proof. There are two cases to consider:

- $t > T_{GST} - d$: At time $t > \ell time(\alpha')$, node i begins the garbage collection. By triggered gossip, node i immediately sends out messages to every node in $world_i$. Node i receives responses from every node in $R_\ell \cup W_\ell$ within $2d$ time, for every ℓ such that $c(\ell)$ is in the $gc.cmap_i$, beginning the propagation phase, which likewise ends a further $2d$ later.
- $t \leq T_{GST} - d$: At time t, node i begins the garbage collection. By occasional gossip, i sends out messages to every node in $world_i$ no later than time T_{GST}. By time $T_{GST} + 2d$, node i receives responses from every node in $R_\ell \cup W_\ell$, for every ℓ such that $c(\ell)$ is in the $gc.cmap_i$, beginning the propagation phase, which likewise ends a further $4d$ later.

□

We can now present the main result of this section which shows that every garbage-collection operation completes within $4d$ time:

Theorem 11.17 *Assume that for some node i, a* gc$(k)_i$ *event occurs at time $t \geq 0$. Assume that i does not fail by time $\max(t, T_{GST}) + 4d$. Then a* gc-ack$(k)_i$ *occurs no later than time $\max(t, T_{GST}) + 4d$.*

Proof. By (strong) induction on the number of gc events in α: assume inductively that if ρ is one of the first $n \geq 0$ gc events in α and that ρ is initiated by node j at time t' and that j does not fail by time $\max(t', T_{GST}) + 4d$, then there is a matching $\mathsf{gc\text{-}ack}_j$ by time $\max(t', T_{GST}) + 4d$.

We examine the inductive step: Consider the $(n+1)^{\text{st}}$ $\mathsf{gc}(*)$ event in α. Let γ be the garbage-collection operation initiated by the gc event; let j be the node that initiates γ, let k be the target of γ and let t_{gc} be the time at which γ begins. If j fails by $\max(t_{gc}, T_{GST}) + 4d$, then the conclusion is vacuously true. Consider the case where j does not fail at or before $\max(t_{gc}, T_{GST}) + 4d$.

Lemma 11.16 shows that proving the following is sufficient: for every configuration $\ell \in \mathit{removal\text{-}set}(\gamma) \cup \{k\}$ there exists a read-quorum R and a write-quorum W of configuration $c(\ell)$ such that no node in $R \cup W$ fails by $\max(t_{gc}, T_{GST}) + 3d$. There are two cases to consider:

- *Case 1:* $t_{gc} \leq T_{GST} + 13d$.
  This follows immediately from configuration viability.

- *Case 2:* $t_{gc} > T_{GST} + 13d$.
  Let α_{pre} be the prefix of α ending with the gc event of γ. Fix some configuration $\ell \in \mathit{removal\text{-}set}(\gamma) \cup \{k\}$.

  We now apply Lemma 11.15: Notice that $\mathit{cmap}(\ell) \in C$ at time t_{gc}, by the choice of ℓ in the $\mathit{removal\text{-}set}(\gamma)$. Also, notice by gc-readiness that j performs a $\mathsf{join\text{-}ack}_j$ no later than time $t_{gc} - (e + 3d)$. Finally, observe that the inductive hypothesis implies immediately that α satisfies the gc-completes hypothesis for t_{gc}, since every garbage-collection operation that begins at a time $< t_{gc}$ is necessarily one of the first n garbage collections in α. Thus we conclude from Lemma 11.15 that there exists a read-quorum R and a write-quorum W of configuration $c(\ell)$ such that no node in $R \cup W$ fails by $\max(t_{gc}, T_{GST}) + 3d$. □

We now present one additional corollary, an unconditional version of Lemma 11.15, which states that as long as any configuration is in use by any cmap, it remains viable:

Corollary 11.18 *Fix a time $t \geq 0$. Assume that for some non-failed node i that performs a* $\mathsf{join\text{-}ack}_i$ *no later than $t - (e + 3d)$, for some $k \geq 0$, $\mathit{cmap}(k)_i \in C$ at time t. Then there exists a read-quorum R and a write-quorum W of configuration $c(k)$ such that no node in $R \cup W$ fails by* $\max(t, T_{GST}) + 3d$.

Proof. Consider the case where $t > T_{GST} + 13d$. Notice that the only condition of Lemma 11.15 that is not assumed here is that α satisfies the gc-completes hypothesis for t. This follows immediately from Theorem 11.17, implying the desired conclusion. Alternatively, if $t \leq \ell\mathit{time}(\alpha') + 13d$, the claim follows immediately from configuration-viability. □

11.4 Reads and writes.

Before presenting the main result of this section, we need one further lemma which shows that every node learns rapidly about a newly produced configuration:

Lemma 11.19 *For a given index ℓ, let π be the first* $\mathsf{recon}(c(\ell), *)$ *event, and let i be the node at which it occurs. Let ϕ be the preceding* $\mathsf{report}(c(\ell))_i$ *event, and assume that ϕ occurs at time t. Then there exists a CMap cm such that: (i) $cm(\ell) \neq \bot$, and (ii) cm is mainstream after* $\max(t, T_{GST}) + 6d$.

Proof. Recon-spacing guarantees that there exists a write-quorum $W \in \mathit{write\text{-}quorums}(c(\ell))$ such that for every node $j \in W$, a $\mathsf{report}(c(\ell))_j$ occurs in α prior to π. By configuration-viability, there exists some read-quorum $R \in \mathit{read\text{-}quorums}(c(\ell))$ such that no node in R fails at or before time $\max(t, T_{GST}) + 5d$. Choose $j \in R \cap W$. Since the report action notifies j of the configuration $c(\ell)$ prior to π, we conclude that after time t, $\mathit{cmap}(\ell)_j \neq \bot$. Let $cm = \mathit{cmap}_j$ after time $\max(t, T_{GST}) + 4d$.

Since j does not fail until after $\max(t, T_{GST}) + 5d$, and j is a member of configuration $c(\ell)$, which was initially proposed no later than time t, we conclude by Lemma 11.3 that cm is mainstream after $\max(t, T_{GST}) + 6d$. □

We now show that every read and write operation terminates within $8d$ time. This theorem is quite similar in form to Theorem 10.7.

Theorem 11.20 *Let $t > T_{GST} + 16d$, and assume a read or write operation starts at time t at some node i. Assume that i performs a $\mathsf{join\text{-}ack}_i$ no later than time $t - (e + 8d)$ and does not fail until the read or write operation completes*[5]*. Then node i completes the read or write operation by time $t + 8d$.*

Proof. Let $c_0, c_1, c_2, \ldots$ denote the infinite sequence of successive configurations decided upon in α; by infinite reconfiguration, this sequence exists. For each $k \geq 0$, let π_k be the first $\mathsf{recon}(c_k, c_{k+1})_*$ event in α, let i_k be the location at which this occurs, and let ϕ_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.)

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 ψ be the read_i, write_i or $\mathsf{query\text{-}fix}_i$ event that begins the phase.

We claim that $time(\psi) > time(\phi_0) + 8d$, that is, that ψ occurs more than $8d$ time after the $\mathsf{report}(0)_{i_0}$ event: We have that $time(\psi) \geq t$, and $t > time(\mathsf{join\text{-}ack}_i) + 8d$, by assumption. Also, $time(\mathsf{join\text{-}ack}_i) \geq time(\mathsf{join\text{-}ack}_{i_0})$. Furthermore, $time(\mathsf{join\text{-}ack}_{i_0}) \geq time(\phi_0)$, 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 $time(\psi) > time(\phi_0) + 8d$.

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

Next, we show that before any further time passes after ψ, $cmap(\ell)_i \neq \perp$ for all $\ell \leq k$. (It is at this point that the proof diverges from that of Theorem 10.7.) Fix any $\ell \leq k$. We apply Lemma 11.19 to conclude that there exists a CMap cm such that: (i) $cm(\ell) \neq \perp$, and (ii) cm is mainstream after $\max(time(\phi_\ell), \ell time(\alpha') + e + d) + 6d$. We next apply Theorem 11.6 to conclude that cm is mainstream after $time(\psi)$. Finally, since i performs a $\mathsf{join\text{-}ack}_i$ at least time $e + 2d$ prior to $time(\psi)$, we conclude that after $time(\psi)$, $cm \leq cmap_i$

Now, by choice of k, we know that $time(\psi) \leq time(\phi_{k+1}) + 8d$. The recon-spacing hypothesis implies that $time(\pi_{k+1})$ (the first recon event that requests the creation of the $(k+2)^{nd}$ configuration) is $> time(\phi_{k+1}) + 12d$. Therefore, for an interval of time of length $> 4d$ after ψ, 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 ψ completes with at most one additional delay (of $2d$) for learning about a new configuration. This yields a total time of at most $4d$ for the phase, as claimed. Finally, Corollary 11.18 shows that the configurations remain viable for sufficiently long. □

12 RAMBO as an Architectural Template

We have presented a specification for RAMBO, a new reconfigurable atomic memory for read/write objects, and have presented and analyzed a new, highly nondeterministic, asynchronous message-passing algorithm that implements RAMBO. Our presentation of the specification and of the algorithm had additional goals of clarity, succinctness, and mathematical elegance. At the same time, we view RAMBO as an architectural template for targeted implementations of the service on a variety of distributed platforms, ranging from networks-of-workstations to mobile networks. In fact several related works substantiate our claims that RAMBO is a practical service that is implementable as a distributed system service, in turn making it suitable as a building block for dynamic distributed applications. In this section we overview selected results that are either motivated by RAMBO or that directly use RAMBO as a point of departure for optimizations and practical implementations.

Long-lived operation of RAMBO service. To make RAMBO service practical for long-lived settings where the size and the number of the messages needs to controlled, Georgiou et al. [23] develop two algorithmic refinements. The first introduces a *leave* protocol that allows nodes to gracefully depart from the RAMBO service, hence

[5]Formally, we assume that i does not fail in α, and then notice that if i fails after the operation terminates, that has no effect on the claim at hand.

reducing the number of, or completely eliminating messages sent to the departed nodes. The second reduces the size of messages by introducing an incremental communication protocol. The two combined modifications are proved correct by showing that the resulting algorithm implements RAMBO. Musial [42, 43] implemented the algorithms on a network-of-stations, experimentally showing the value of these modifications.

Restricting gossiping patterns and enabling operation restarts. To further reduce the volume of gossip messages in RAMBO, Gramoli et al. [28] constrain the gossip pattern so that gossip messages are sent only by the nodes that (locally) believe that they have the most recent configuration. To address the side-effect of some nodes potentially becoming out-of-date due to reduced gossip (nodes may become out-of-date in in RAMBO as well), the modified algorithm allows for a non-deterministic operation restarts. The modified algorithm is proved to be implementing RAMBO, and the experimental implementation [42, 43] is used to illustrate th advantage of the approach. In practice, non-deterministic operation restart is most effectively replaced by a heuristic decision based on local observations, such as the duration of an operation in progress. Of course any such heuristic preserves correctness.

Implementing a complete shared memory service. The RAMBO service is specified for a single object, with complete shared memory implemented by composing multiple instances of the algorithm. In practical system implementations this may result in significant communication overhead. Georgiou et al. [24] developed a variation of RAMBO that introduces the notion of *domains*, collections of related objects that share configurations, thus eliminating much of the overhead incurred by the shared memory obtained through composition of RAMBO instances. A networks-of-workstations experimental implementation is also provided. Since the specification of the new service includes domains, the proof of correctness is achieved by adapting the proof we presented here to that service.

Indirect learning in the absence of all-to-all connecticity. Our RAMBO algorithm assumes that either all nodes are connected by direct links or that an underlying network layer provides transparent all-to-all connectivity. Assuming this may be unfeasible or prohibitively expensive to implement in dynamic networks, such ad hoc mobile settings. Konwar et al. [35] develop an approach to implementing RAMBO service where all-to-all gossip is replaced by an *indirect learning* protocol for information dissemination. The indirect learning scheme is used to improve the liveness of the service in the settings with uncertain connectivity. The algorithm is proved to implement RAMBO service. The authors examine deployment strategies for which indirect learning leads to an improvement in communication costs,

Integrated reconfiguration and garbage collection. The RAMBO algorithm decouples reconfiguration (the issuance of new configurations) from the garbage collection of obsolete configurations. We have discussed the benefits of this approach in this paper. In some settings it is beneficial to tightly integrate reconfiguration with garbage collection. Doing so in the settings where there is a concern for imminent failure of the current configuration may reduce the latency of removing such configuration. Gramoli [27] and Chockler et al. [12] integrate reconfiguration with garbage collection by “opening” the external consensus service, such as that used by RAMBO, and combining it with the removal of the old configuration. For this purpose they use Paxos algorithm [36] as the starting point, and the RAMBO configuration upgrade protocol. The resulting reconfiguration protocol reduces the latency of garbage collection as compared to RAMBO. The drawback of this approach is that it ties reconfiguration to a specific consensus algorithm. In contrast, the loose coupling in RAMBO allows one to implement specialized *Recon* services that are most suitable for particular deployment scenarios as we discuss next.

Dynamic atomic memory in sensor networks. Beal et al. [10] developed an implementation of the RAMBO framework in the context of a wireless ad hoc sensor network. In this context, configurations are defined with respect to a specific geographic region: every sensor within the geographic region is a member of the configuration, and each quorum consists of a majority of the members. Sensors can store and retrieve data via RAMBO read and write operations, and the geographic region can migrate via reconfiguration. In addition, reconfiguration can be used to incorporate newly deployed sensors, and to retire failed sensors. An additional challenge was to *efficiently* implement the necessary communication (presented in this paper as point-to-point channels) in the context of a wireless network that supports one-to-many communication.

Dynamic atomic memory in mobile ad hoc networks. Dolev et al. [17] developed a new approach for implementing atomic read/write shared memory in mobile ad hoc networks where the individual stationary locations constituing the members of a fixed number of quorum configurations are implemented by mobile devices. Motivated in part by RAMBO, their work specializes RAMBO algorithms in two ways. (1) In RAMBO the first (query) phase of write operations serves to establish a timestamp that is higher than any timestamps of the previously completed writes. If a global time service is available, then taking a snapshot of the global time value obviates the need for the first phase in write operations. (2) The full-fledged consensus service is necessary for reconfiguration in RAMBO only when the universe of possible configurations is unknown. When the set of possible configurations is small and known in advance, a much simpler algorithm suffices. The resulting approach, called GeoQuorums, yields an algorithm that efficiently implements read and write operations in a highly dynamic, mobile network.

Distributed enterprise disk arrays. Finally, we note that Hewlett-Packard recently used a variation of RAMBO in their implementation of a "federated array of bricks" (FAB), a distributed enterprise disk array [3, 46].

13 Conclusions

We presented a specification for a new reconfigurable atomic memory for read/write objects; we call it RAMBO. We also presented an analyzed an asynchronous message-passing algorithm that implements RAMBO. The algorithm is highly dynamic and we show its correctness (safety) in the presence of any pattern of asynchrony and failures. Some of our liveness properties depend on assumed message delay bounds and limited failure. Our analysis is very conservative and there are many possibilities for using weaker assumptions. Our RAMBO framework, viewed as an architectural template, allows for a variety of optimizations and implementations in specific target networks, from networks-of-workstations to ad hoc mobile networks to enterprise disk arrays. We anticipate that the approach defined and presented in this paper will continue to influence follow on research and practical implementations of atomic memory services in dynamic systems.

Acknowledgments. The authors thank Ken Birman, Alan Demers, Rui Fan, Butler Lampson, and Peter Musial for helpful and insightful 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] Albrecht, Jeannie R. and Saito, Yasushi. RAMBO for Dummies. HPL-2005-39, Hewlett-Packard External Tech. Report. 20050304, 5 pages, 2005.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Jacob Beal, Seth Gilbert, "RamboNodes for the Metropolitan Ad Hoc Network", in *Proc. of DIWANS Workshop, International Conference on Dependable Systems and Networks*, Florence, Italy, 2004.

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

[12] Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M. Musial, Alexander A. Shvartsman. Reconfigurable Distributed Storage for Dynamic Networks. Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS'05), Springer-Verlag, LNCS 3974, pages 214–219, 2005.

[13] 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.

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

[15] 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.

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

[17] Shlomi Dolev, Seth Gilbert, Nancy A. Lynch, Alexander A. Shvartsman, Jennifer L. Welch. GeoQuorums: implementing atomic memory in mobile ad hoc networks. Distributed Computing. Volume 18 , Issue 2 , Pages 125-155, December, 2005.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] Chryssis Georgiou and Peter M. Musial and Alexander A. Shvartsman, Long-lived Rambo: Trading knowledge for communication, Theor. Comput. Sci., volume 383, number 1, 2007, pages 59–85.

[24] Chryssis Georgiou and Peter M. Musial and Alexander A. Shvartsman, Developing a Consistent Domain-Oriented Distributed Object Service, Fourth IEEE International Symposium on Network Computing and Applications (NCA 2005), pages 149–158.

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

[26] 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

[27] Vincent Gramoli, RAMBO III: Speeding-up the reconfiguration of an atomic memory service in dynamic distributed system. Thesis. Universit Paris Sud - Orsay, France, September, 2004.

[28] Vincent C. Gramoli and Peter M. Musial and Alexander A. Shvartsman, Operation Liveness and Gossip Management in a Dynamic Distributed Atomic Data Service, Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems (PDCS), 2005, pages 206–211.

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

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

[31] 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.

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

[33] 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).

[34] 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.

[35] Konwar, Kishori M. Musial, Peter M. Nicolaou, Nicolas C. Shvartsman, Alex A. Implementing Atomic Data through Indirect Learning in Dynamic Networks. Sixth IEEE International Symposium on Network Computing and Applications (NCA), Publication Date: 12-14 July 2007 pages 223–230, 2007.

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

[37] 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.

[38] 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.

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

[40] 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.

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

[42] Piotr M. Musial, *From High Level Specification to Executable Code: Specification, Refinement, and Implementation of a Survivable and Consistent Data Service for Dynamic Networks*. Ph.D. Dissertation, Computer Science and Eng., University of Connecticut, Storrs, 2007.

[43] P.M. Musial and A.A. Shvartsman. Implementing a Reconfigurable Atomic Memory Service for Dynamic Networks. In Proceedings of 18'th International Parallel and Distributed Symposium – FTPDS WS. Santa Fe, NM, pages 208b, 2004.

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

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

[46] Yasushi Saito, Svend Frlund, Alistair C. Veitch, Arif Merchant, Susan Spence: FAB: building distributed enterprise disk arrays from commodity components. Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, pages 48–58.

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

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