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

Seth Gilbert
EPFL, Lausanne, Switzerland
seth.gilbert@epfl.ch

Nancy A. Lynch
MIT, Cambridge, USA
lynch@theory.lcs.mit.edu

Alexander A. Shvartsman
U. of Connecticut,
Storrs, CT, USA
aas@cse.uconn.edu

Abstract

In this paper, we present RAMBO, an algorithm for emulating a read/write distributed shared memory in a dynamic, rapidly changing environment. RAMBO provides a highly reliable, highly available service, even as participants join, leave, and fail. In fact, the entire set of participants may change during an execution, as the initial devices depart and are replaced by a new set of devices. Even so, RAMBO ensures that data stored in the distributed shared memory remains available and consistent.

There are two basic techniques used by RAMBO to tolerate dynamic changes. Over short intervals of time, replication suffices to provide fault-tolerance. While some devices may fail and leave, the data remains available at other replicas. Over longer intervals of time, RAMBO copes with changing participants via *reconfiguration*, which incorporates newly joined devices while excluding devices that have departed or failed. The main novelty of RAMBO lies in the combination of an efficient reconfiguration mechanism with a quorum-based replication strategy for read/write shared memory.

The RAMBO algorithm can tolerate a wide variety of aberrant behavior, including lost and delayed messages, participants with unsynchronized clocks, and, more generally, arbitrary asynchrony. Despite such behavior, RAMBO guarantees that its data is stored consistency. We analyze the performance of RAMBO during periods when the system is relatively well-behaved: messages are delivered in a timely fashion, reconfiguration is not too frequent, etc. We show that in these circumstances, read and write operations are efficient, completing in at most eight message delays.

Keywords: dynamic distributed systems, atomic register, distributed shared memory, fault-tolerance, reconfigurable, eventual synchrony

[*] 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. This work was supported in part by the NSF ITR Grant CCR-0121277. The work of the second author was additionally supported by the NSF Grant 9804665, and the work of the third author was additionally supported in part by the NSF Grants 9984778, 9988304, and 0311368.

1 Introduction

In this paper, we present RAMBO, an algorithm for emulating a read/write distributed shared memory in a dynamic, constantly-changing setting. (RAMBO stands for "Reconfigurable Atomic Memory for Basic Objects.) Read/write shared memory is a fundamental and long-studied primitive for distributed algorithms, allowing each device to store and retrieve information. Key properties of a distributed shared memory include consistency, availability, and fault tolerance.

We are particularly interested in dynamic environments in which new participants may continually join the system, old participants may continually leave the system, and active participants may fail. For example, during an execution, the entire set of participating devices may change, as the initial participants depart and a new set of devices arrive to take their place. Many modern networks exhibit this type of dynamic behavior. For example, in peer-to-peer networks, devices are continually joining and leaving; peers often remain in the network only long enough to retrieve the data they require. Or, as another example, consider mobile networks; devices are constantly on the move, resulting a continual change in participants. In these types of networks, the set of participants is rarely stable for very long. Especially in such a volatile setting, it is of paramount importance that devices can reliably share data.

The most fundamental technique for achieving fault-tolerance is *replication*. The RAMBO protocol replicates the shared data at participating device, thus maximizing the likelihood that the data survives. When the data is modified, the replicas must be updated in order to maintain the consistency of the system. RAMBO relies on classical *quorum-based* techniques to implement consistent read and write operations. Specifically, the algorithm uses *configurations*, each of which consists of a set of *members* (i.e., replicas), plus sets of *read-quorums* and *write-quorums*. (A quorum is defined as a set of devices.) The key requirement is that every read-quorum intersects every write-quorum. Each operation retrieves information from some read-quorum and propagates it to some write-quorum, ensuring that any two operations that use the same configuration access some common device, i.e., the device in the intersection of the two quorums. In this way, RAMBO ensures that the data is accessed and modified in a consistent fashion.

This quorum-based replication strategy ensures the availability of data over the short-term, as long as not too many devices fail or depart. As long as all the devices in at least one read-quorum and one write-quorum remain, we say that the configuration is *viable*, and the data remains available.

Eventually, however, configuration viability may be compromised due to continued changes in the set of participants. Thus, RAMBO supports *reconfiguration*, that is, replacing one set of members/quorums with an updated set of members/quorums. In the process, the algorithm propagates data from the old members to the new members, and allows devices that are no longer members of the new configuration to safely leave the system. This changeover has no effect on ongoing data-access operations, which may continue to store and retrieve the shared data.

1.1 Algorithmic Overview

In this section, we present a high-level overview of the RAMBO algorithm. RAMBO consists of three sub-protocols: (1) $Joiner$, a simple protocol for joining the system; (2) $Reader\text{-}Writer$, the main protocol that implements read and write operations, along with garbage-collecting old obsolete configurations; and (3) $Recon$, a protocol for responding to reconfiguration requests. We now describe how each of these components operates.

Joiner. The first component, $Joiner$, is quite straightforward. When a new device wants to participate in the system, it notifies the *Joiner* component of RAMBO that it wants to join, and it initializes RAMBO with a set of *seed* devices that have already joined the system. We refer to this set of seed devices as the *initial world view*. The $Joiner$ component contacts these devices in the initial world view and retrieves the information necessary for the new device to participate.

Reader-Writer. The second component, $Reader\text{-}Writer$, is responsible for executing the read and write operations, as well as for performance "garbage collection."

A read or write operation is initiated by a client that wants to retrieve or modify an elementin the distributed shared memory. Each operation executes in the context of one or more *configurations*. At any given time, there is always at least one active configuration, and sometimes more than one active configuration (when a new configuration has been installed, but the old configuration has not yet been removed). Each read and write operation must use *all* active configurations. (Coordinating the interaction between concurrent operations and reconfiguration is one of the key challenges in maintaing consistency while ensuring continued availability.)

Each configuration specifies a set of members, i.e., replicas, that each hold a copy of the shared data. Each operation consists of two phases: a *query* phase, in which information is retrieved from one (or more) read-quorums, and a *propagate* phase, in which information is sent to one (or more) write-quorums; for a write operation, the information propagated is the value being written; for a read operation, it is the value that is being returned.

The *Reader-Writer* component has a secondary purpose: garbage-collecting old configurations. In fact, a notable aspect of RAMBO is that reconfiguration is split into two parts: producing new configurations, which is the responsibility of the *Recon* component, and removing old configurations, which is the responsibility of the *Reader-Writer* component. The decoupling of garbage-collection from the production of new configurations is a key feature of RAMBO which has several benefits. First, it allows read and write operations to proceed concurrently with ongoing reconfigurations. This is especially important when reconfiguration is slow (due to asynchrony in the system); it is critical in systems that have real-time requirements. Second, it allows the *Recon* component complete flexibility in producing configurations, i.e., there is no restriction requiring configurations to overlap in any way. Since garbage collection propagates information forward from one configuration to the next, the *Reader-Writer* ensures that consistency is maintained, regardless.

The garbage-collection operation proceeds much like a read or write operation in that it consists of two phases: in the first phase, the initiator of the *gc* operation contacts a set of read-quorums and write-quorums from the old configuration, collecting information on the current state of the system; in the second phase, it contacts a write-quorum of the new configuration, propagating necessary information to the new participants. When this is completed, the old configuration can be safely removed.

Recon. The third component, *Recon*, is responsible for managing reconfiguration. When a participant wants to reconfigure the system, it proposes a new configuration via a recon request. The main goal of the *Recon* component is to respond to these requests, selecting one of the proposed configurations. The end result is a sequence of configurations, each to be installed in the specified order. At the heart of the *Recon* component is a *distributed consensus* service that allows the participants to agree on configurations. Consensus can be implemented using a version of the Paxos algorithm [38].

1.2 RAMBO Highlights

In this section, we describe the guarantees of RAMBO and discuss its performance. In Section 2, we discuss how these features differ from other existing approaches.

Safety Guarantees. RAMBO guarantees atomicity (i.e., linearizability), regardless of network behavior or system timing anomolies (i.e., asynchrony). Messages may be lost, or arbitrarily delayed; clocks at different devices may be out-of-synch and may measure time at different rates. Despite these types of non-ideal behavior, every execution of RAMBO is guaranteed to be atomic, meaning that it is operationally equivalent to an execution of a centralized shared memory. The safety guarantees of RAMBO are captured by Theorem 6.1 in Section 6.

Performance Guarantees. We next consider RAMBO's performance. In order to analyze the latency of read and write operations, we make some further assumptions regarding RAMBO's behavior, for example, requiring it to regularly send gossip messages to other participants. We also restrict the manner in which the join protocol is initialized: recall that each *join* request includes a seed set of participates that have already joined the system; we assume that these sets overlap sufficiently such that within a bounded period of time, every node that has joined the system is aware of every other node that has joined the system.

There are a few further key assumptions, specifically related to reconfiguration and the viability of the proposed configurations. In particular, we assume that every configuration remains viable until sufficiently long after the next new configuration is installed. This is clearly a necessary restriction: if a non-viable configuration (consisting, perhaps, of too many failed devices) is installed, then it is impossible for the system to make progress. We also assume that reconfigurations are not initiated too frequently, and that every proposed configuration consists of devices that have already completed the join protocol.

We show that when these assumptions hold, read and write operations are efficient. With regards to network and timing behavior, we consider two separate cases. In Section 9, we assume that the network is always well-behaved, delivering messages reliably within d time. Moreover, we assume that every process has a clock that measures time accurately (with respect to real time). Under these synchrony assumptions, we show that old configurations are rapidly garbage-collected when new configurations are installed (see Lemma 9.6), and that every read and write operation completes within $8d$ time (see Theorem 9.7).

We also consider the case where the network satisfies these synchrony requirements only during certain intervals of the execution: during some periods of time, the network may be well-behaved, while during other periods of time, the network may lose and delay messages. We show that during periods of synchrony, RAMBO stabilizes soon after synchrony resumes, guaranteeing as before that every read and write operation completes within $8d$ time (see Theorem 10.20).

Key Aspects of RAMBO. A notable feature of RAMBO is that any configuration may be installed at any time. In particular, there is no requirement that quorums in a new configuration intersect quorums from an old configurations. In fact, a new configuration may include an entirely disjoint set of members from all prior configurations. RAMBO achieves this by depending on *garbage collection* to propagate information from one configuration to the next (before removing the old configuration); by contrast, several prior algorithms depend on the fact that configurations intersect to ensure consistency.

This capacity to install any configuration provides the client with great flexibility when proposing new configurations. Often, the primary goal of the client is to choose configurations that will remain viable for as long as possible; this task is simplified by allowing the client to include exactly the set of participants that it believes will remain extant for as long as possible. (While the selection of configurations is outside the scope of this paper, we discuss briefly in Section 8 the issues associated with choosing good configurations.)

Another notable feature of RAMBO is that the main read/write functionality is only loosely coupled with the reconfiguration process. Read and write operations continue, even as a reconfiguration may be in progress. This provides two benefits. First, even very slow reconfigurations do not delay read and write operations. When the network is unstable and network latencies fluctuate, the process of agreeing on a new configuration may be slow. Despite these delays, read and write operations can continue. Second, by allowing read and write operations to always make progress, RAMBO guarantees more predictable performance: a time-critical application cannot be delayed too much by unexpected reconfiguration.

1.3 Roadmap

In this section, we provide an overview of the rest of the paper, which is organized as follows:

- Sections 2–3 contain introductory material and other necessary preliminaries:

  In Section 2, we discuss some interesting related research, and some other approaches for coping with highly-dynamic rapidly changing environments. In Section 3 we present further preliminaries on the system model and define some basic data types.

- Sections 4–7 present the RAMBO algorithm, and show that it guarantees consistent read and write operations:

  In Section 4 we provide a formal specification for a global service that implements a reconfigurable atomic shared memory. We also present a specification for a *Recon* service that manages reconfiguration; this service is used as a subcomponent of our algorithm. Each of these specifications includes a set of *well-formedness* requirements that the user of the service must follow, as well as a set of *safety* guarantees that the service promises to deliver.

  In Section 5 we present two of the RAMBO sub-protocols: *Reader-Writer* and *Joiner*. We present pseudocode, using the I/O automata formalism, and discuss some of the issues that arise. In Section 6, we show that these components, when combined with any *Recon* service, guarantee atomic operations. In Section 7, we present our implementation of *Recon* and show that it satisfies the specified properties. Together, Sections 5 and 7 provide a complete instantiation of the RAMBO protocol.

- Sections 8–10 analyze the performance of RAMBO:

  We analyze performance *conditionally*, based on certain failure and timing assumptions that are described in Section 8. In Section 9 we study the case where the network is well-behaved throughout the execution, while in Section 10 we consider the case where the network is *eventually* well-behaved. In both cases, we show that read and write operations complete within $8d$, where d is the maximum message latency.

- Section 11 summarizes the paper, and discuss some interesting open questions.

2 Related Work and Other Approaches

In this section, we discuss other research that is relevant to RAMBO. In the first part, Section 2.1, we focus on several important papers that introduced key techniques related to replication, quorum systems, and reconfiguration. In the second part, Section 2.2, we focus on alternate techniques for implementing a dynamic distributed shared memory. Specifically, we describe how replicated state machines and group communication systems can both be used to implement a distributed shared memory, and how these approaches differ from RAMBO. In the third part, Section 2.3, we present an overview of some of the research that has been carried out subsequent to the original publication of RAMBO.

2.1 Replication, Quorum Systems, and Reconfiguration

We begin by discussing some of the early work on quorum-based algorithms. We then proceed to discuss dyanmic quorum systems, group communication services, and other reconfigurable systems.

Quorum Systems. Upfal and Wigderson demonstrated the first general scheme for emulating shared-memory in a message-passing system [59]. This scheme involves replicating data across a set of replicas, and accessing a majority of the replicas for each request. Attiya, Bar-Noy and Dolev generalized this approach, developing a majority-based emulation of atomic registers that uses bounded time-stamps [9]. Their algorithm introduced the two-phase paradigm in which information is gathered in the first phase from a majority of processors, and information is propagated in the second phase to a majority of processors. It is this two-phase paradigm that is at the heart of RAMBO's implementation of read and write operations.

Quorum systems [24] are a generalization of simple majorities. A *quorum system* (also called a *coterie*) is a collection of sets such that any two sets, called *quorums*, intersect [21]. (In the case of a majority quorum system, every set that consists of a majority of devices is a quorum.) A further refinement of this approach divides quorums into read-quorums and write-quorums, such that any read-quorum intersects any write-quorum; sometimes there is the additional requirement that any two write-quorums intersect. Each configuration used by RAMBO consists of such a quorum system.

Quorum systems have been widely used to ensure consistent coordination in a fault-prone distributed setting. Quorums have been used to implement distributed mutual exclusion [21] and data replication protocols [16, 30]. Quorums have also been used in the context of transaction-style synchronization [12]). Other replication techniques that use quorums include [1,2,4,11,27]. An additional level of fault-tolerance in quorum-based approaches can be achieved using the Byzantine quorum approach [6, 46].

There has been a significant body of research that attempts to quantify the fault-tolerance provided by various quorum systems. For example, there has been extensive research studying the trade-off between *load*, *availability*, and *probe complexity* in quorums systems (see, e.g., [51–53]). Quorum systems have been studied under probabilistic usage patterns, and in the context of real network data, in an attempt to evaluate the likelihood that a quorum system remains viable in a real-life setting (see, e.g., [8, 41, 52, 56]). While the choice of configurations is outside the scope of this paper, the techniques developed in these papers may be quite relevant to choosing good configurations that will ensure sufficient viability.

Dynamic Quorum Systems. Previous research has considered the problem of adapting quorum systems dynamically as an execution progresses. One such approach is referred to as "dynamic voting" [31, 32, 42]. Each of these schemes places restrictions on the choice of quorum systems, often preventing such a system from being used in a truly dynamic environment. They also often rely on locking (or mutual exclusion) during reconfiguration, thus reducing the fault tolerance. For example, the approach in [32] relies on locking and requires (approximately) that at least a majority of all the processors in some previously updated quorum remain alive. The approach in [42] does not rely on locking, but requires at least a predefined number of processors to always be alive. The online quorum adaptation of [11] assumes the use of Sanders [58] mutual exclusion algorithm, which again relies on locking.

Other significant recent research on dynamic quorum systems includes [3, 50]. These papers each present a specific quorum system and examine its performance (i.e., load, availability, and probe complexity) in the context of a dynamic network. They then show how the quorum system can be adapted over time as nodes join and leave. In [50] they show how nodes that gracefully leave the system can hand off data to their neighbors in a dynamic Voronoi diagram, thus allowing for continued availability. In [3], by contrast, they consider a probabilistic quorum system, and show how to evolve the data as nodes join and leave to ensure that data remains available. This is accomplished by maintaining a dynamic de Bruijn graph. Both of these quorum systems have significant potential for use in dynamic systems.

Group Communication Services. Group communication services (GCS) provide another approach for implementing a distributed shared memory in a dynamic network. This can be done, for example, by implementing a global totally ordered broadcast service on top of a view-synchronous GCS [19] using techniques of Amir, Dolev, Keidar, Melliar-Smith and Moser [7, 34, 35].

De Prisco, Fekete, Lynch, and Shvartsman [54] introduce a group communication service that can tolerate a dynamically changing environment. They introduce the idea of "primary configurations" and defined a dynamic primary configuration group communication service. Using this group communication service as a building block, they show how to implement a distributed shared memory in a dynamic environment using a version of the algorithm of Attiya, Bar-Noy, and Dolev [9] within each configuration (much as is done in RAMBO). Unlike RAMBO, this earlier work restricted the choice of a new configuration, requiring certain intersection properties between new and old configurations. RAMBO, by contrast, allows complete flexibility in the choice of a new configuration.

In many ways, GCS-based approaches share many of the disadvantages associated with the "replicated state machine approach" which we discuss in more detail in Section 2.2. These approaches (often) involve a tight coupling between the reconfiguration mechanism and the main common-case operation; in general, operations are delayed whenever a reconfiguration occurs. These approaches (often) involve agreeing on the order of each operation, which can also be slow.

Single Reconfigurer Approaches. Lynch and Shvartsman [18, 44] also consider a *single reconfigurer* approach to the problem of adapting to a dynamic environment. In this model, a single, distinguished device is responsible for initiating all reconfiguration requests. This approach, however, is inherently not fault tolerant, in that the failure of the single reconfigurer disables all future reconfiguration. By contrast, in RAMBO, any member of the latest configuration may propose a new configuration, avoiding any single point of failure. Another difference is that in [18, 44], garbage-collection of old configurations is tightly coupled to the introduction of a new configuration. Our approach in this paper allows garbage-collection of old configurations to be carried out in the background, concurrently with other operations. A final difference is that in [18, 44], 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 unviable. By contrast, in RAMBO, information about configurations is gossiped periodically, in the background, which permits all participants to learn about new configurations and garbage-collect old configurations. Despite these differences, many of the ideas that appear in RAMBO were pioneered in these early papers.

2.2 Replicated State Machine Approach

While RAMBO relies on consensus for choosing new configurations, it is possible to use a consensus service directly to implement a distributed shared memory. This is often referred to as the *replicated state machine* approach, as originally proposed by Lamport [37]. It was further developed in the context of Paxos [38, 55], and has since become, perhaps, the standard technique for implementing fault-tolerance distributed services.

The basic idea underlying the replicated state machine paradigm is as follows. Assume a set of participating devices is attempting to implement a sepcific service. Each participant maintains a replica of the basic state associated with that service. Whenever a client submits an operation to the service, the participants run an agreement protocol, i.e., *consensus*, in order to ensure that the replicated state is updated consistently at all the participants. The key requirement of the agreement protocol is that the participants agree on an ordering of all operations so that each participant performs local updates in the same order.

In the context of a shared memory implementation, there are two types of operations: read requests and write requests. Thus, when implemented using the replicated state machine approach, the participants agree on the ordering of these operations. (Further optimization can allow some portion of these agreement instances to execute in parallel.) Since each read and write operation is ordered with respect to every other operation, it is immediately clear how to implement a read operation: each read request returns the value written by the most recent preceding write operation in the ordering.

In this case, reconfiguration can be integrated directly into the same framework: reconfiguration requessts are also ordered in the same way with respect to all other operations. When the participants agree that the next operation should be a recon operation, the set of participants is modified as specified by the reconfiguration request. In many ways, this approach is quite elegant, as it relies on only a single mechanism (i.e., consensus). It avoids complicated interactions among concurrent operations (as may occur in RAMBO) by enforcing a strict ordering of all operations. Recent work [14, 39, 40] has shown how to achieve very efficient operation in good circumstances, i.e., when a leader has been elected that does not leave or fail, and when message delays are bounded. In this case, each operation can complete in only two message delays, if there are sufficiently many correct process, and three message delays, otherwise. It seems clear that in a relatively stable environment, such as a corporate data center, a replicated state machine solution has many advantages.

RAMBO, by contrast, decouples read and write operations from reconfiguration. From a theoretic perspective, the advantage here is clear: in an asynchronous, fault-prone system, it is impossible to guarantee that an instance of consensus will terminate [20]. Thus, it is possible for an adversary to delay any operation from completing in a replicated state machine for an arbitrarily long period of time (even without any of the devices actually failing). In RAMBO, by contrast, read and write operations can always terminate. No matter what asynchrony is inflicted on the system, read and write operations cannot be delayed forever. (In both RAMBO and the replicated state machine approaches, configuration viability is a pre-requisite for any operations to terminate.)

From a practical perspective, the non-termination of consensus is rarely a problem. Even so, RAMBO may be preferable in more dynamic, less stable situations. Replicated state machine solutions tend to be most efficient when a single leader can be selected. For example, Paxos [38], which is today one of the most common implementations of distributed consensus, makes progress only when all participants agree on a single (correct) leader. This is often accomplished via *failure detectors* and *timeout* mechanisms that attempt to guess when a particular leader candidate is available. When the network is less well-behaved, or the participation more dynamic, it is plausible that agreeing on a stable leader may be quite slow. For example, consider the behavior of a peer-to-peer service that is distributed over a wide-area network. Different participants on different continents may have different views of the system, and latencies between pairs of nodes may differ significantly. Over such a wide-area network, message loss is relatively common, and sometimes messages can be significantly delayed. In such circumstances, it is unclear that a stable leader can be readily and rapidly selected. Similarly, in a wireless mobile network, communication is remarkably unreliable due to collisions and electromagnetic interference; the highly dynamic participation may make choosing a stable long-term leader quite difficult.

RAMBO, by contrast, can continue to respond to read and write requests, despite such unpredictable network behavior, as it depends only the quorum system remaining viable. It is certainly plausible to imagine situations (particularly those involving intermittent connectivity) in which a quorum system may remain viable, but a stable leader may be hard to find. In RAMBO, consensus is used only to determine the ordering of configurations. Reconfiguration may be (inevitably) slow, if the network is unpredictable or the participation dynamic. However by decoupling read and write operations from reconfiguration, RAMBO minimizes the harm caused by a dynamic and unpredictable environment.

Thus, in the end, it seems that both RAMBO and the replicated state machine approaches have their merits, and it remains to examine experimentally how these approaches work in practice.

2.3 Extensions of RAMBO

Since the original preliminary RAMBO papers appeared [25, 26, 45], RAMBO has formed the basis for much ongoing research. Some of this research has focused on tailoring RAMBO to specific distributed platforms, ranging from networks-of-workstations to mobile networks. Other research has used RAMBO as a building block for higher-level applications, and it has been optimized for improved performance in certain domains.

In many ways, it is the highly non-deterministic and modular nature of RAMBO that makes it so adaptable to various optimizations. A given implementation of RAMBO has significant flexibility in when to send messages, how many messages to send, what to include in messages, whether to combine messages, etc. An implementation can substitute a more involved *Joiner* protocol, or a different *Recon* service. In this way, we see the RAMBO algorithm presented in this paper as a template for implementing a reconfigurable service in a highly dynamic environment.

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 [10, 15, 22, 23, 28, 29, 36, 47, 48, 57].

- **Long-lived operation of RAMBO service.** To make the 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 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 [47, 48] implemented the algorithms on a network-of-workstations, 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. [29] 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 RAMBO as well), the modified algorithm allows for non-deterministic operation restart. The modified algorithm is shown to implement RAMBO, and the experimental implementation [47, 48] is used to illustrate the advantages of this approach. In practice, non-deterministic operation restarts are 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. [22] 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 connectivity.** 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. [36] 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 [28] and Chockler et al. [15] 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 the Paxos algorithm [38] as the starting point, and the RAMBO garbage-collection 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, and cause increased delays under some circumstances. 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 a peer-to-peer network.** Muthitacharoen et al. [49] developed an implementation of the RAMBO framework, known as Etna, in the context of a peer-to-peer network. Etna guaranteed fault-tolerant, mutable data in a distributed hash table (DHT), using an optimized variant of RAMBO to maintain the data consistently despite continual changes in the underlying network.

- **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 constituting 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 time stamp that is higher than any time stamps 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 [5, 57]. In this paper, we have adopted some of the presentations suggestions found in [5] (and elsewhere), specifically related to the out-of-order installation of configurations.

3 Preliminaries

In this section, we introduce several fundamental concepts and definitions that we will use throughout the paper. We begin by discussing the basic computational devices that populate our system. Next, we discuss the shared memory being emulated. Third, we discuss the underlying communication network. Fourth, we introduce the idea of configurations. Finally, we conclude with some notational defintions.

Fault-Prone Devices. The system consists of a set of devices communicating via an all-to-all asynchronous message-passing network. Let I be a totally-ordered (possibly infinite) set of identifiers, each of which represents a device in the system. We refer to each device as a *node*. We occasionally refer to a node that has joined the system as a *participant*.

Each node executes a program consisting of two components: a *client* component, that is specified by the user of the distributed shared memory, and the RAMBO component, which executes the RAMBO algorithm for emulating a distributed shared memory. The client component issues read and write requests, while the RAMBO components responds to these requests.

Nodes may fail by crashing, i.e., stopping without warning. When this occurs, the program—including both the client components and the RAMBO components—takes no further steps.

We define T to be a set of *tags*. That is, $T = \mathbb{N} \times I$. These tags are used to order the values written to the system.

Shared Memory Objects. The goal of a distributed shared memory is to provide access to a set of read/write objects. Let X be a set of object identifiers, each of which represents one such object. For each object $x \in X$, let V_x be the set of values that object x may take on; let $(v_0)_x$ be the initial value of object x. We also define $(i_0)_x$ to be the initial *creator* of object x, i.e., the node that is initially responsible for the object. After the first reconfiguration, $(i_0)_x$ can delegate this responsibility to a broader set of nodes.

Communication Network. Processes communicate via point to point channels $Channel_{x,i,j}$, one for each $x \in X$, $i, j \in I$ (including the case where $i = j$). Let M be the set of messages that can be sent over one of these channels. We assume that every message used by the RAMBO protocol is included in M.

The channel $Channel_{x,i,j}$ is accessed using $\mathsf{send}(m)_{x,i,j}$ input actions, by which a sender at node i submits message $m \in M$ associated with object x to the channel, and $\mathsf{receive}(m)_{x,i,j}$ output actions, by which a receiver at node j receives $m \in M$. 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. Formally, we model the channel as a multiset. A send adds the message to the multiset, and any message in the multiset may be delivered via a receive. Note, however, that a receive does not remove the message.

Configurations. Let C be a set of identifiers which we refer to as *configuration identifiers*. Each identifier $c \in C$ is associated with a configuration which consists of three components:

- $members(c)$, a finite subset of I.
- $read\text{-}quorums(c)$, a set of finite subsets of $members(c)$.
- $write\text{-}quorums(c)$, a set of finite subsets of $members(c)$.

We assume that for every $c \in C$, for every $R \in read\text{-}quorums(c)$, and for every $W \in write\text{-}quorums(c)$, $R \cap W \neq \emptyset$. That is, every read-quorum intersections every write-quorum.

For each $x \in X$, we define $(c_0)_x$ to be the *initial configuration identifier* of x. We assume that $members((c_0)_x) = \{(i_0)_x\}$. That is, the initial configuration for object x has only a single member, who is the creator of x.

Partial Orders. 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 $a \in A$.

4 Specifications

In Section 4.1, we provide a detailed specification for the behavior of a read/write distributed shared memory. The main goal of this paper is to present an algorithm that satisfies this specification.

A key subcomponent of the RAMBO algorithm is a reconfiguration service that manages the problem of responding to reconfiguration requests. In Section 4.2, we provide a specification for such a reconfiguration service.

4.1 Reconfigurable Atomic Memory Specification

In this section, we give a specification for a reconfigurable atomic memory service. This specification consists of an external signature (i.e., an interface) plus a set of traces that embody the safety properties. No liveness properties are included in the specification. We provide a conditional performance analysis in Sections 9 and 10. The definition of a distributed reconfigurable atomic memory can be found in Definition 4.4.

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

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

Figure 1: External signature for a reconfigurable atomic memory. There are four main request/response pairs: join/join-ack, read/read-ack, write/write-ack, and recon/recon-ack.

Signature

The external signature for a reconfigurable atomic memory service appears in Figure 1. It consists of four basic operations: join, read, write, and recon, each of which returns an acknowledgment. It also accepts a fail input, and produces a report output. We now proceed in more detail. For the rest of this section, consider $i \in I$ to be some node in the system.

Node i issues a request to join the system for a particular object x by performing a join$(\mathit{rambo}, J)_{x,i}$ input action. The set J represents the client's best guess at a set of processes that have already joined the system for x. We refer to this set J as the *initial world view* of i. If the join attempt is successful, the RAMBO service responds with a join-ack$(\mathit{rambo})_{x,i}$ response.

Node i initiates a read or write operation by requesting a read$_i$ or a write$_i$ (respectively), which the RAMBO service acknowledges with a read-ack$_i$ response or a write-ack$_i$ response (respectively).

Node i initiates a reconfiguration by requesting a recon$_i$, which is acknowledged with a recon-ack$_i$ response. Notice that when a reconfiguration is acknowledged, this does not imply that the configuration was installed; it simply means that the request has been processed. New configurations are reported by RAMBO via report$_i$ outputs. Thus a node can determine whether its reconfiguration request was successful by observing whether the proposed configuration is reported.

Finally, a crash at node 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.

Safety Properties

We now define the safety guarantees, i.e., the properties that are to be guaranteed by every execution. Under the assumption that the client requests are well-formed, a reconfigurable atomic memory service guarantees that the responses are also well-formed, and that the read and write operations satisfy atomic consistency. In order to define these guarantees, we specify a set of traces that capture exactly the guaranteed behavior.

We now proceed in more detail. We first specify what it means for requests to be well-formed. In particular, we require that a node i issues no further requests after it fails, that a node i issues a join request before initiating read and write operations, that node i not begin a new operation until it has received acknowledgments from all previous operations, and that configurations are unique.

Definition 4.1 (Reconfigurable Atomic Memory Request Well-Formedness) For every object $x \in X$, node $i \in I$, configurations $c, c' \in C$:

1. *Failures:* After a fail$_i$ event, there are no further join$(\mathit{rambo}, *)_{x,i}$, read$_{x,i}$, write$(*)_{x,i}$, or recon$(*, *)_{x,i}$ requests.
2. *Joining:* The client at i issues at most one join$(\mathit{rambo}, *)_{x,i}$ request. Any read$_{x,i}$, write$(*)_{x,i}$, or recon$(*, *)_{x,i}$ event is preceded by a join-ack$(\mathit{rambo})_{x,i}$ event.
3. *Acknowledgments:* The client at i does not issue a new read$_{x,i}$ request or write$_{x,i}$ requst until there has been a read-ack or write-ack for any previous read or write request. The client at i does not issue a new recon$_{x,i}$ request until there has been a recon-ack for any previous reconfiguration request.
4. *Configuration Uniqueness:* The client at i issues at most one recon$(*, c)_{x,*}$ request. This says that configuration identifiers 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.
5. *Configuration Validity:* If a recon$(c, c')_{x,i}$ rquest occurs, then it is preceded by: (i) a report$(c)_{x,i}$ event, and (ii) a join-ack$(\mathit{rambo})_{x,j}$ event for every $j \in \mathit{members}(c')$. This says that the client at i can request reconfiguration from c to c' only if i has previously received a report confirming that configuration c has been installed, and only if all the members of c' have already joined. Notice that i may have to communicate with the members of c' to ascertain that they are ready to participate in a new configuration.

When the requests are well-formed, we require that the responses also be well-formed:

Definition 4.2 (Reconfigurable Atomic Memory Response Well-Formedness) For every object $x \in X$, and node $i \in I$:

1. *Failures:* After a fail$_i$ event, there are no further join-ack$(rambo)_{x,i}$, read-ack$(*)_{x,i}$, write-ack$_{x,i}$, recon-ack$()_{x,i}$, or report$(*)_{x,i}$ outputs.
2. *Acknowledgments:* Any join-ack$(rambo)_{x,i}$, read-ack$(*)_{x,i}$, write-ack$_{x,i}$, or recon-ack$()_{x,i}$ outputs has a preceding join$(rambo, *)_{x,i}$, read$_{x,i}$, write$(*)_{x,i}$, or recon$(*, *)_{x,i}$ request (respectively) with no intervening request or response for x and i.

We also require that the read and write operations satisfy *atomicity*.

Definition 4.3 (Atomicity) For every object $x \in X$: If all read and write operations complete in an execution, 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 requests 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 $(v_0)_x$.

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 [43][1].

We can now specify precisely what it means for an algorithm to implement a reconfigurable atomic memory:

Definition 4.4 (Reconfigurable Atomic Memory) We say that an algorithm A implements a reconfigurable atomic memory if it has the external signature found in Figure 1 and if every trace β of A satisfies the following:

> If requests in β are well-formed (Definition 4.1), then responses are well-formed (Definition 4.2) and operations in β are atomic (Definition 4.3).

4.2 Reconfiguration Service Specification

In this section, we present the specification for a generic reconfiguration service. The main goal of a reconfiguration service is to respond to reconfiguration requests and produce a (totally ordered) sequence of configurations. A reconfiguration service will be used as part of the RAMBO protocol (and we show in Section 7 how to implement it). We proceed by describing its an external signature, along with a set of traces that define its safety guarantees. The reconfiguration service specification can be found in Definition 4.8.

Input:	Output:
join$(recon)_i$, $i \in I$	join-ack$(recon)_i$, $i \in I$
recon$(c, c')_i$, $c, c' \in C$, $i \in members(c)$	recon-ack$()_i$, $i \in I$
request-config$(k)_i$, $k \in \mathbb{N}^+$, $i \in I$	report$(c)_i$, $c \in C$, $i \in I$
fail$_i$, $i \in I$	new-config$(c, k)_i$, $c \in C$, $k \in \mathbb{N}^+$, $i \in I$

Figure 2: External signature for a reconfiguration service. A reconfiguration service has three main request/response pairs: join/join-ack, recon/recon-ack, and request-config/new-config.

Signature

The interface for the reconfiguration service appears in Figure 2. Let node $i \in I$ be a node in the system. Node i requests to join the reconfiguration service by performing a join$(recon)_i$ request. The service acknowledges this with a corresponding join-ack$_i$ response. The client initiates a reconfiguration using a recon$_i$ request, which is acknowledged with a recon-ack$_i$ response. When node i is ready for the k^{th} configuration in the reconfiguration sequence, it issues a request-config(k) request. When the k^{th} configuration has been agreed upon, the reconfiguration service responds with a new-config$(c, k)_i$, announcing configuration c at node i.

The service also announces new configurations to the client, producing a report$_i$ output to provide an update when a new configuration is installed. Crashes are modeled using fail input actions.

[1] Lemma 13.16 of [43] is presented for a setting with only finitely many nodes, whereas we consider infinitely many nodes. However, nothing in Lemma 13.16 or its proof depends on the finiteness of the set of nodes, so the result carries over immediately to our setting. In addition, 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 as well.

Safety Properties

Now we define the set of traces describing $Recon$'s safety properties. Again, these are defined in terms of environment well-formedness requirements and service guarantees. The well-formedness requirements are as follows:

Definition 4.5 (Recon Request Well-Formedness) For every node $i \in I$, configuration $c, c' \in C$:

1. *Failures:* After a fail_i event, there are no further $\mathsf{join}(recon)_i$ or $\mathsf{recon}(*,*)_i$ requests.
2. *Joining:* At most one $\mathsf{join}(recon)_i$ request occurs. Any $\mathsf{recon}(*,*)_i$ request is preceded by a $\mathsf{join\text{-}ack}(recon)_i$ response.
3. *Acknowledgments:* Any $\mathsf{recon}(*,*)_i$ request is preceded by an -ack response for any preceding $\mathsf{recon}(*,*)_i$ event.
4. *Configuration Uniqueness:* For every c, at most one $\mathsf{recon}(*,c)_*$ event occurs.
5. *Configuration Validity:* For every c, c', and i, if a $\mathsf{recon}(c,c')_i$ request occurs, then it is preceded by: (i) a $\mathsf{report}(c)_i$ output, and (ii) a $\mathsf{join\text{-}ack}(recon)_j$ for every $j \in members(c')$.

We next describe the well-formedness guarantees of the reconfiguration service:

Definition 4.6 (Recon Response Well-Formedness) For every node $iinI$:

1. *Failures:* After a fail_i event, there are no further $\mathsf{join\text{-}ack}(recon)_i$, $\mathsf{recon\text{-}ack}(*)_i$, $\mathsf{report}(*)_i$, or $\mathsf{new\text{-}config}(*,*)_i$ responses.
2. *Acknowledgments:* Any $\mathsf{join\text{-}ack}(recon)_i$ or $\mathsf{recon\text{-}ack}(c)_i$ response has a preceding $\mathsf{join}(recon)_i$ or recon_i request (respectively) with no intervening request or response action for i.
3. *Configuration Requests:* Any $\mathsf{new\text{-}config}(*,k)_i$ is preceded by a $\mathsf{request\text{-}config}(k)_i$.

A reconfiguration service also guarantees that configurations are produced consistently. That is, for every node i, the reconfiguration service outputs an ordered set of configurations; the configurations are exactly those proposed, and every node is notified about an identical sequence of configurations.

Definition 4.7 (Configuration Consistency) For every node $i \in I$, configurations $c, c' \in C$, and index k:

1. *Agreement:* If $\mathsf{new\text{-}config}(c,k)_i$ and $\mathsf{new\text{-}config}(c',k)_j$ both occur, then $c = c'$. Thus, no disagreement arises about the k^{th} configuration identifier, for any k.
2. *Validity:* If $\mathsf{new\text{-}config}(c,k)_i$ occurs, then it is preceded by a $\mathsf{recon}(*,c)_{i'}$ request for some i'. Thus, any configuration identifier that is announced was previously requested.
3. *No duplication:* If $\mathsf{new\text{-}config}(c,k)_i$ and $\mathsf{new\text{-}config}(c,k')_{i'}$ both occur, then $k = k'$. Thus, the same configuration identifier cannot be assigned to two different positions in the sequence of configuration identifiers.

We can now specify precisely what it means to implement a reconfiguration service:

Definition 4.8 (Reconfiguration Service) We say that an algorithm A implements a reconfiguration service if it has the external signature described in Figure 2 and if every trace β of A satisfies the following:

> If β satisfies Recon Request Well-Formedness (Definition 4.5), then it satisfies Recon Response Well-Formedness (Definition 4.6) and Configuration Consistency (Definition 4.7).

5 The RAMBO Algorithm

In this section, we describe the RAMBO algorithm. The RAMBO algorithm includes three components: (1) $Joiner$, which handles the joining of new participants; (2) $Reader\text{-}Writer$, which handles reading, writing, and garbage-collecting old configurations; and (3) $Recon$, which produces new configurations.

In this section, we describe the first two components, postponing the description of $Recon$ until Section 7. For the purpose of this section, we simply assume that some generic reconfiguration service is available that satisfies Definition 4.8.

Notice that we can consider each object $x \in X$ separately. The overall shared memory emulation can be described, formally, as the composition of a separate implementation for each x. Therefore, throughout the rest of the paper, we fix a particular $x \in X$, 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.

Signature:

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

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

State:

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

Transitions:

```
Input join$(rambo, J)_i$
Effect:
    if $status = idle$ then
        $status \leftarrow joining$
        $hints \leftarrow J$

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

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

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

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

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

Figure 3: $Joiner_i$: This component of the RAMBO protocol handles join requests. Each join request includes an initial world view. The main responsibility of the join protocol is to contact at least one node in the initial world view.

5.1 Joiner automata

The goal of the Joiner automata is to handle join requests. The signature, state and pseudocode of $Joiner_i$, for node $i \in I$, appear in Figure 3.

When $Joiner_i$ receives a join$(rambo, J)$ request from its environment, it carries out a simple protocol. First, it sets its $status$ to $joining$, and sends $join$ messages to the processes in J, i.e., those in the initial world view (via send$(join)_{i,*}$ actions). It sends these messages with the hope that at least some nodes in J are already participating, and so can help in the attempt to join. These messages are received by the $Reader\text{-}Writer$ automaton by nodes in J, which then send a response to the $Reader\text{-}Writer$ component at i.

At the same time, it submits join requests to the local $Reader\text{-}Writer$ and $Recon$ components and waits for acknowledgments for these requests. The $Reader\text{-}Writer$ component completes its join protocol when it receives a response from nodes in J. The $Recon$ service completes its own join protocol independently. When both the $Reader\text{-}Writer$ and $Recon$ components have completed their join protocol, the $Joiner$ automaton performs a join-ack, setting its status to $active$.

5.2 Reader-Writer automata

The main part of the RAMBO algorithm is the reader-writer algorithm, which handles read and write requests. Each read or write operation takes place in the context of one or more configurations. The reader-writer protocol also handles the garbage-collection of older configurations, which ensures that later read and write operations need not use them.

Signature and state

The signature and state of $Reader\text{-}Writer_i$ appear in Figure 4. The $Reader\text{-}Writer$ signature includes an interface to process read and write requests: read, read-ack, write, and write-ack. It also includes an interface for communicating with the $Joiner$ automaton, from which it receives a join(rw) request and returns a join-ack(rw) response. And it includes an

Signature:

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

Output:
read-ack$(v)_i$, $v \in V$
write-ack$_i$
join-ack$(rw)_i$
request-config$(k)_i$, $k \in \mathbb{N}^+$
send$(m)_{i,j}$, $m \in M, j \in I$

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

State:

$status \in \{idle, joining, active, failed\}$, initially $idle$
$world$, a finite subset of I, initially $\emptyset$
$value \in V$, initially v_0
$tag \in T$, initially $(0, i_0)$
$cmap : \mathbb{N} \rightarrow C_{\pm}$, a configuration map, initially:
$cmap(0) = c_0$,
$cmap(k) = \bot$ for $k \geq 1$
$pnum\text{-}local \in \mathbb{N}$, initially 0
$pnum\text{-}vector$, a mapping from I to $\mathbb{N}$, initially everywhere 0

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

gc, a record with fields:
$phase \in \{idle, query, prop\}$, initially $idle$
$pnum \in \mathbb{N}$
$cmap : \mathbb{N} \rightarrow C_{\pm}$, a configuration map
acc, a finite subset of I
$target \in N$

Figure 4: $Reader\text{-}Writer_i$: Signature and state for the *Reader-Writer* component of the RAMBO algorithm. The *Reader-Writer* component is responsible for handling read and write requests, as well as for garbage-collecting old configurations.

interface for communicating with the reconfiguration service, from which it receives $new\text{-}config$ reports whenever a new configuration is selected. Finally, it includes $send$ and $recv$ actions for communicating with other nodes. Notice that one of the recv actions is dedicated to receiving join-related messages.

We now describe the state maintained by $Reader\text{-}Writer$. The $status$ variable keeps track of the progress as the node 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 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 nodes that have attempted to join the system. Gossip messages are sent regularly to every node in $world$.

The $value$ variable contains the current value of the local replica of x, and tag holds the associated tag. Every value written to the object x has a unique tag associated with it, and these tags are used to determine the order in which values have been written.

The $cmap$ variable is a "configuration map" that contains information about configurations. A *configuration map* is a function that maps each index k to one of three types: $\bot$, a configuration c, or $\pm$. If $cmap(k) = \bot$, it means that $Reader\text{-}Writer_i$ has not yet learned about the k^{th} configuration. If $cmap(k) = c$, 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.

Throughout the execution, we ensure that the $cmap$ always has the following form: a finite (possibly zero length) sequence of indices mapped to $\pm$, followed by at least one, and possibly more, indices mapped to C, followed by an infinite number of indices mapped to $\bot$. That is, such a *cmap* is of the form:

$$\langle \underbrace{\pm,\ \pm,\ \ldots,\ \pm,}_{\geq 0}\ \underbrace{c,\ c',\ \ldots,\ c'',}_{\geq 1}\ \underbrace{\bot,\ \bot,\ \ldots\ldots}_{\infty} \rangle$$

When a *cmap* satisfies this pattern, we say that it is Usable. When there is some k such that $\mathit{cmap}(k) = c$, we say that configuration c is active. When $\mathit{Reader\text{-}Writer}_i$ processes a read or write operation, it uses all active configurations.

We define the following two functions that combine two different configuration maps. First, the function *update* simply merges two *cmap*s, taking the "more recent" element from each *cmap*. The *update* function takes two configuration maps as input, and returns a new configuration map; it is defined in a pointwise fashion. For all indices k:

$$\mathit{update}(\mathit{cm1}, \mathit{cm2})(k) = \begin{cases} \pm & \text{if } \mathit{cm1}(k) = \pm \text{ or } \mathit{cm2}(k) = \pm \\ \mathit{cm1}(k) & \text{if } \mathit{cm1}(k) \in C \\ \mathit{cm2}(k) & \text{if } \mathit{cm2}(k) \in C \\ \bot & \text{otherwise} \end{cases}$$

The *extend* function takes two configuration maps, and includes all the configurations available in both. Unlike the *update* function, it includes a configuration that is in one *cmap*, even if it is garbage collected in the other. Again, *extend* takes two configuration maps as input, and returns a new configuration map; it is defined in a pointwise fashion. For all indices k:

$$\mathit{extend}(\mathit{cm1}, \mathit{cm2})(k) = \begin{cases} \mathit{cm2}(k) & \text{if } \mathit{cm1}(k) = \bot \\ \mathit{cm1}(k) & \text{otherwise} \end{cases}$$

The *pnum-local* variable and *pnum-vector* array are used to implement a handshake that identifies "recent" messages[2]. $\mathit{Reader\text{-}Writer}_i$ uses *pnum-local* to count the total number of "phases" that node i has initiated; a phase can be a part of a read, write, or garbage-collection operation. For every j, $\mathit{Reader\text{-}Writer}_i$ records in $\mathit{pnum\text{-}vector}(j)$ the largest phase that j has started, to the best of node i's knowledge. A message m from i to j is deemed "recent" by j if i knows about j's current phase, i.e., if $\mathit{pnum\text{-}vector}(j)_i \geq \mathit{pnum\text{-}local}_j$. This implies that i has received a message from j that was sent after j began the new phase and was received prior to i sending the message to j.

Finally, two records, op_i and gc_i, are used to maintain information about read, write, and garbage-collection operation that were initiated by node i and are still in progress. Each of these records includes a *phase*, to keep track of the status of the operation (e.g., *idle*, in a *query* phase, in a *prop* phase, or *done*). They also maintain the *pnum* associated with the ongoing phase (if some operation is in progress), along with an operation-specific *cmap*. The set *acc* contains a set of clients that have sent responses since the phase began. (As described above, the phase numbers are used to verify which messages are sufficiently recent.) In addition, the *op* field maintains a *value* associated with the operation (for example, the value being written), and the *gc* field maintains the *target* of the garbage-collection operation, i.e., the smallest configuration that will remain after the operation completes.

Pseudocode

The state transitions are presented in three figures: Figure 5 presents the pseudocode pertaining to joining the system, learning new configurations, and failing (or leaving). Figure 6 presents the pseudocode pertaining to propagating information and performing read/write operations. Figure 7 presents the pseudocode pertaining to garbage-collection. We divide the discussion into four parts: (1) joining; (2) read and write operations; (3) information propagation; (4) configuration management.

Joining. The pseudocode associated with joining is presented in Figure 5. When a $\mathsf{join}(\mathit{rw})_i$ request occurs and $\mathit{status} = \mathit{idle}$, node i begins joining the system. If i is the object's creator, i.e. if $i = i_0$, then *status* is immediately set to *active*, which means that $\mathit{Reader\text{-}Writer}_i$ is ready for full participation in the protocol. Otherwise, *status* becomes *joining*, which means that $\mathit{Reader\text{-}Writer}_i$ is receptive to inputs but not ready to perform any locally initiated actions. In either case, $\mathit{Reader\text{-}Writer}_i$ records itself as a member of its own view of the *world*.

From this point on, whenever a $\mathsf{recv}(\mathit{join})_{j,i}$ event occurs at node i, $\mathit{Reader\text{-}Writer}_i$ adds j to its *world*. (Recall that these *join* messages are sent by *Joiner* automata, not *Reader-Writer* automata.) Information is propagated lazily to members of the *world*, so no response is sent here.

The process of joining is completed as soon as $\mathit{Reader\text{-}Writer}_i$ receives a message from another node that has already joined. (The code for this appears in the recv transition definition in Figure 6.) At this point, process i has acquired enough information to begin participating fully. Thus, *status* is set to *active*, after which process i performs a $\mathsf{join\text{-}ack}(\mathit{rw})$.

[2] Together, *pnum-local* and *pnum-vector* implement something akin to a *vector clock* [37] in that they are used to determine some notion of causality; however, the usage is simpler and the guarantees weaker than a vector clock.

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

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

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

```
Output $\mathsf{request\text{-}config}(k)_i$
Precondition:
    $status = active$
    $\forall k' < k : cmap(k) \neq \bot$
    $cmap(k) = \bot$
Effect:
    none

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

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

Figure 5: $Reader\text{-}Writer_i$: Joining, reconfiguration, and failing.

Read and write operations. The pseudocode associated with read and write operations is in Figure 6. A read or write operation is performed in two phases: a query phase and a propagation phase. In each phase, $Reader\text{-}Writer_i$ contacts a set of quorums, one for each active configuration, to obtain recent $value$, tag, and $cmap$ information. This information is obtained by sending and receiving messages in the background, as described below.

Each phase takes place in the context of a set of active configurations, i.e., the configurations that are available in the configuration map $cmap$. When $Reader\text{-}Writer_i$ starts either a query phase or a propagation phase of a read or write, it sets $op.cmap$ to the configuration map $cmap$; this specifies which configurations are to be used to conduct the phase. For example, during a read request, the query phase begins (i.e., $op.phase$ is set to $query$) and $op.cmap$ is set to $cmap$. As the phase progresses, newly discovered configurations are added to $op.cmap$.

Node i begins a query phase by initializing the op structure. (This occurs when a read or write is requested, or when a query phase restarts during a recv, which is discussed below.) Throughout the phase, node i collects information from nodes, attempting to find a quorum. Whenever node i receives new information from a node j, it adds it to $op.acc$. The phase terminates when a "query fixed point" is reached. This happens when $Reader\text{-}Writer_i$ determines that it has received recent responses from some read-quorum of each configuration in its current $op.cmap$. That is, for every configuration $c \in op.cmap_i$, there exists a read-quorum R such that $R \subseteq op.acc_i$. When this occurs, a $\mathsf{query\text{-}fix}_i$ event occurs. We refer to this as a "fixed point" since it captures some notion of convergence: even as i is collecting messages from nodes in the requisite quorums, it may also be learning about new configurations that require it to contact more quorums. Eventually, when the fixed point is reached, node i has contacted all the configurations that are known to any of the configurations which it has contacted.

Let v and t denote process i's $value$ and tag at the query fixed point, respectively. 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. This implies that v is at least the most recent value of any operation that preceded the ongoing read operation.

If the operation is a read operation, then value v is an appropriate value to return to the 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 (along with the associated value). Again, the information is propagated in the background, and $op.cmap$ is managed in the same way. 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 $op.cmap$. At this point, a $prop\text{-}fix_i$ event occurs, meaning that the tag value of each process in each node of the write-quorums is at least as great as t.

Consider, now, a write operation that has completed its query phase. Suppose t, process i's tag at the query fixed point, is of the form (n, j). Then $Reader\text{-}Writer_i$ increments the tag, defining the tag for its write operation to be the pair $(n + 1, i)$; that is, it increments the counter, and uses its own identifier to ensure that the new tag is unique. $Reader\text{-}Writer_i$ sets its local tag to $(n + 1, i)$ and its $value$ to v, the value it is currently writing. Then it performs a propagation phase. As before, 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. The propagation phase is over when the same propagation fixed point condition is satisfied as for the read operation, i.e., there is some write quorum for every active configuration that has received a message from i and sent a response.

```
Input $\mathsf{read}_i$
Effect:
    if $status \notin \{idle, failed\}$ then
        $pnum\text{-}local \leftarrow pnum\text{-}local + 1$
        $op.pnum \leftarrow pnum\text{-}local$
        $op.type \leftarrow read$
        $op.phase \leftarrow query$
        $op.cmap \leftarrow cmap$
        $op.acc \leftarrow \emptyset$

Input $\mathsf{write}(v)_i$
Effect:
    if $status \notin \{idle, failed\}$ then
        $pnum\text{-}local \leftarrow pnum\text{-}local + 1$
        $op.pnum \leftarrow pnum\text{-}local$
        $op.type \leftarrow write$
        $op.phase \leftarrow query$
        $op.cmap \leftarrow cmap$
        $op.acc \leftarrow \emptyset$
        $op.value \leftarrow v$

Input $\mathsf{recv}(\langle world', v, t, cm, snder\text{-}phase, rcver\text{-}phase\rangle)_{j,i}$
Effect:
    if $status \notin \{idle, failed\}$ then
        $status \leftarrow active$
        $world \leftarrow world \cup world'$
        if $t > tag$ then $(value, tag) \leftarrow (v, t)$
        $cmap \leftarrow update(cmap, cm)$
        $pnum\text{-}vector(j) \leftarrow \max(pnum\text{-}vector(j), snder\text{-}phase)$
        if $op.phase \in \{query, prop\}$ and $rcver\text{-}phase \geq op.pnum$ then
            $op.cmap \leftarrow extend(op.cmap, cm)$
            if $op.cmap \in Usable$ then
                $op.acc \leftarrow op.acc \cup \{j\}$
            else
                $pnum\text{-}local \leftarrow pnum\text{-}local + 1$
                $op.acc \leftarrow \emptyset$
                $op.cmap \leftarrow cmap$
        else if $gc.phase \in \{query, prop\}$ and $rcver\text{-}phase \geq gc.pnum$
            $gc.acc \leftarrow gc.acc \cup \{j\}$

Output $\mathsf{send}(\langle world', v, t, cm, snder\text{-}phase, rcver\text{-}phase\rangle)_{i,j}$
Precondition:
    $status = active$
    $j \in world$
    $\langle world', v, t, cm, snder\text{-}phase, rcver\text{-}phase\rangle =$
        $\langle world, value, tag, cmap, pnum\text{-}local, pnum\text{-}vector(j)\rangle$
Effect:
    none
```

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

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

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

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

Figure 6: $Reader\text{-}Writer_i$: Read/write transitions

Information propagation. Information is propagated between $Reader\text{-}Writer$ processes in the background. 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, $pnum\text{-}local$, and the latest known phase number of the receiver, from the $pnum\text{-}vector$ 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. In follow-up papers, Georgiou et al. [23] and Gramoli et al. [29] study the problem of reducing message size, sending gossip "incrementally", and reducing gossip frequency.

When $Reader\text{-}Writer_i$ receives a message (see the recv transition, Figure 6), the first step it takes is to set its $status$ to $active$, if it has not already done so. It then adds any new information about the world contained in the message its local $world$ set. Next, it compares the incoming tag t to its own tag. It t is strictly greater thatn tag, that means that the incoming message contains 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. $Reader\text{-}Writer_i$ then updates its own configuration map, $cmap$, with the information in the incoming configuration map, cm, using the $update$ operator to merge the two configuration maps. In this way, node i learns about any new configurations that were known to the send, and learns about any configurations that have been garbage collected. Next,

$\mathit{Reader\text{-}Writer}_i$ updates its $\mathit{pnum\text{-}vector}(j)$ component for the sender j to reflect new information about the phase number of the sender, which appears in the $\mathit{snder\text{-}phase}$ components of the message.

The rest of the recv transition is dedicated to processing read, write, and garbage-collection operations. In each phase of each operation, the $\mathit{Reader\text{-}Writer}$ collects messages from a quorum of each active configuration. The set $op.acc$ stores the set of nodes from which i has received a message since the operation began. More specifically, node i needs to verify that a message is sufficiently "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. This is accomplished by checking if the incoming phase number $\mathit{rcver\text{-}phase}$ is at least as large as the current operation phase number (*op.pnum*), which ensures that node j has received a message from i since the operation began. If the message is recent, then it is used to update the record associated with the operation.

The first step, if an operation is ongoing and the message is sufficiently recent, is to update the $cmap$ associated with that operation (which is stored in *op.cmap*). In this case, we use the $extend$ function to merge the new configuration map with the old configuration map. This ensures that the extended configuration map includes both every configuration that was in the old *op.cmap* and every configuration in the newly received cm.

Notice that node i must contact a quorum from every new configuration that it learns about in this manner. Even though it may have already contacted quorums from the "old" set of configurations, there is no guarantee that this information was propagated to the new configuration: garbage collection operations are used to transfer information from old configurations to new configurations, but i cannot determine whether or not there is a concurrent garbage collection, and if there is, whether it has come before or after i's access to the old configuration. Thus, to be sure that it has up-to-date information, i must also contact the newer configuration.

It is also important that i continue to access an old configuration, even if it discovers that it has already been garbage-collected. To see why, consider the following example: assume that node i begins a read operation with both configurations c and c' active, where c' is a newly installed configuration that has not yet received the most up-to-date tag/value; node i then receives messages from a quorum of nodes in c', but messages to/from c are lost; next, node j garbage-collects configuration j, propagating the most recent tag/value from c to c'. Notice that at this point, there is no guarantee that i has received the most recent tag/value, as it has only received messages from c', which may have been out-of-date at the time. Thus, if i proceeds without contacting configuration c, then i would risk returning out-of-date data. (Another alternative is to re-start the operation instead.)

After extending the $op.cmap$, the $\mathit{Reader\text{-}Writer}_i$ needs to verify that the $op.cmap$ is still in an appropriate form, i.e., it is still Usable. Recall that we maintain the invariant that both $op.cmap$ and the cm received in the message are Usable. However, after extending the configuration map, it may no longer be Usable. Consider the case where the two (usable) configuration maps are as follows:

$$\begin{aligned} op.cmap &= \langle \pm, \pm, c_1, c_2, \bot, \bot, \bot, \bot, \bot, \bot, \ldots \rangle \\ cm &= \langle \pm, \pm, \pm, \pm, \pm, \pm, \pm, c_3, \bot, \bot, \ldots \rangle \end{aligned}$$

Notice that when these two configurations maps are combined using the $extend$ function, we end up with:

$$\langle \pm, \pm, c_2, c_2, \pm, \pm, \pm, c_3, \bot, \bot, \ldots \rangle$$

The resulting configuration map is not usable. This can in fact happen when node i gets too far out-of-date, i.e., all messages to and from i are lost for a longer period of time, during which multiple configurations are installed and removed. When this happens, an ongoing read/write operation must be restarted. Effectively, this is for the same reasons described above: the read/write operation must use every newly installed configuration; however, some of the newly installed configurations have already been removed, and hence are not available. Thus, there is no choice but for the operation to restart. If the operation needs to restart, than the local phase number is incremented, the set $op.acc$ is reset to $\emptyset$, and the $op.cmap$ is reset to $cmap$, the most recently known configuration map. Otherwise, if the operation is not reset, then the sender of the message is added to $op.acc$.

As in the case of a read or write operation, the garbage collection operation also requires $\mathit{Reader\text{-}Writer}_i$ to collect a set of recent messages from other nodes. Thus, the last step in the the recv transition is to check if the message is recent with respect to $gc.pnum$, and if so, add the sender to $gc.acc$. In this case, there is no need to update any of the other gc fields, and there is never a need to restart the operation; if the configuration in question has already been removed, then the garbage-collection operation can simply terminate.

New configurations and garbage collection. When a node has learned about all configurations with index smaller than k, then it informs the Recon service that it is ready for configuration k via a request-config(k) request. (This ensures that the Recon service informs the $\mathit{Reader\text{-}Writer}$ of configurations in order, ensuring that there are no gaps in the configuration map.)

Eventually, the $Recon$ service may inform the $Reader\text{-}Writer_i$ of a new configuration via a new-config(c, k) event. When $Reader\text{-}Writer_i$ hears about a new configuration identifier via a new-config input action, it simply records it in its $cmap$ and updates the $op.cmap$ (in case an operation is active). The pseudocode associated with interactions with the $Recon$ service is in Figure 5.

From time to time, configuration identifiers get garbage-collected at i. It is during this garbage-collection process that information is propagated from one configuration to the next, and that enables the installation of disjoint configurations, i.e., configurations that have no overlapping quorums. The pseudocode associated with configuration management is in Figure 7.

There are two situations in which $Reader\text{-}Writer_i$ may remove a configuration c. First, $Reader\text{-}Writer_i$ can remove c if it ever hears that another process has already garbage-collected it. For example, assume that at node i $cmap(k-1) = c$, and that node i then receives a gossip message where $cm(k-1) = \pm$; in this case, node i can set $cmap(k) = \pm$, and we say that configuration c has been garbage collected.

The second where $Reader\text{-}Writer_i$ may remove configuration c is when it itself initiates a garbage-collection operation. For example, assume that $cmap(k-1) = c$ at node i, and assume that i has learned about a new configuration, i.e., $cmap(k) \neq \bot$. In this case, $Reader\text{-}Writer_i$ may initiate a garbage-collection of all configurations with index less than k (see the gc transition in Figure 7). Garbage collection is a two-phase operation with a structure similar to the read and write operations. In addition, 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 smaller than k, that is, every configuration such that for $\ell < k$, $gc.cmap(\ell) \in C$. The query phase accomplishes two tasks. First, $Reader\text{-}Writer_i$ ensures that sufficient information is conveyed to the processes in the read-quorums and write-quorums of other active configurations with index smaller than k. In particular, all these processes in "old" configurations learn about the existence of configuration k, and also learn that all configurations smaller than k are being garbage-collected. We refer loosely to the fact that they know about configuration k as the "forwarding pointer" condition—if a node j that has such a forwarding pointer is contacted at a later time by someone who is trying to access a quorum of configuration $k-1$, node j is able to inform it about the existence of the newer configuration k. This ensures that operations can continue to make progress, even as configurations are removed.

The second purpose of the query phase is to collect recent tag and $value$ information from the old configurations. This ensures that, by the end of the query phase, $Reader\text{-}Writer_i$ has received a value as recent as any written prior to the garbage collection beginning. More precisely, it'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 $Reader\text{-}Writer_i$ for the query phase.

The propagation phase of a garbage-collection operation is straightforward: $Reader\text{-}Writer_i$ ensures that the tag and value are propagated to a write-quorum of the new configuration k. This ensures that the new configuration is appropriately informed of preceding operations, and that future operations can safely rely on configuration k.

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, $Reader\text{-}Writer_i$ does not start with some set of configurations in $op.cmap$ and extend it as it learns about more configurations. Rather, $Reader\text{-}Writer_i$ 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 $Reader\text{-}Writer_i$ is carrying out a garbage-collection operation, it may discover that someone else has already garbage-collected all the configurations smaller than k; it discovers this by observing that $cmap(\ell) = \pm$ for all $\ell < k$. When this happens, $Reader\text{-}Writer_i$ may simply terminate its operation.

5.3 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 that implements a reconfiguration service, as described in Definition 4.8—with all the actions that are not part of the reconfigurable atomic memory interface hidden.

6 Proof of Safety

In this section, we show that our implementation $\mathcal{S}$ satisfies the safety guarantees of RAMBO, as given in Section 4, assuming the environment safety assumptions. That is, we prove the following theorem:

Theorem 6.1 RAMBO *implements a reconfigurable atomic memory, as in Definition 4.4. That is, let β be a trace of the system $\mathcal{S}$. If requests in β are well-formed (as in Definition 4.1), then responses are well-formed (as in Definition 4.2) and operations in β are atomic (as in Definition 4.3). That is, $\mathcal{S}$ is a Reconfigurable Atomic Memory (Definition 4.4).*

```
Internal $\mathsf{gc}(k)_i$
Precondition:
    $status = active$
    $gc.phase = idle$
    $cmap(k) \in C$
    $cmap(k-1) \in C$
    $\forall \ell < k : cmap(\ell) \neq \bot$
Effect:
    $pnum\text{-}local \leftarrow pnum\text{-}local + 1$
    $gc.phase \leftarrow query$
    $gc.pnum \leftarrow pnum\text{-}local$
    $gc.cmap \leftarrow cmap$
    $gc.acc \leftarrow \emptyset$
    $gc.target \leftarrow k$

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

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

Internal $\mathsf{gc\text{-}ack}(k)_i$
Precondition:
    $status = active$
    $gc.target = k$
    $\forall \ell < k :\ cmap(\ell) = \pm$
Effect:
    $gc.phase \leftarrow idle$
```

Figure 7: $Reader\text{-}Writer_i$: Garbage-collection transitions

The proof of well-formedness is straightforward based on inspection of the code, and based on the properties of the $Recon$ service (which ensure that configurations are installed sequentially). The rest of this section is devoted to the proof of atomicity. We consider a trace β of $\mathcal{S}$ that satisfies the well-formedness assumptions and in which all read and write operations complete. We show the existence of a partial order on operations in β that satisfies the conditions listed in Definition 4.3.

6.1 Proof Overview

We begin with an overview of the basic structure of the proof. Recall that each read and write operation has a tag associated with it. For a write operation, this tag is chosen by the node that is executing the operation. For a read operation, this tag is the largest tag discovered during the first phase of the operation. These tags define a partial order in which operations are ordered based on their tags; operations with the same tag are unordered with respect to each other. We show that this partial order satisfies the requisite properties needed to ensure atomicity.

The key claim regarding this partial order is the following: if one operation completes before a second operation begins, then the second operation is not ordered before the first, i.e., the tag associated with the second operation is at least as large as the tag associated with the first operation. If the first operation is a write operation, then the first operation is ordered before the second operation, i.e., the tag of the first operation is strictly less than the tag of the second operation. (The remaining three properties are relatively straightforward.)

If two operations share a configuration, then this claim is easily proven. Consider two such operations, the second of which begins after the first completes. During the propagation phase, the first operation conveys its tag and value to a write quorum. During the query phase, the second operation retrieves tags from a read quorum. Since these two quorums intersect, we can be sure that the second operation find a tag at least as large as the first, implying the desired ordering of operations.

A more careful argument is needed when the two operations do *not* share a configurations. For example, it is possible that all the configurations accessed by the first operation have been garbage-collected (and removed) prior to the beginning of the second operation. In this case, the two operations may access an entirely disjoint set of quorums. The key to the proof, then, is showing that the tag is propagated from the first operation to the second operation by the intervening garbage-collection operations. To this end, there are three key claims.

Claim 1. The first claim is that the tag is correctly propagated from the first operation to some configuration. During the propagation phase, the tag associated with a read/write operation is sent to every configuration that is used by the operation.

Some of these configurations may have already been garbage-collected prior to the operation beginning; some of these configurations may be in the process of being garbage-collect concurrently. We need to show that the tag is propagated to a configuration that is still active, i.e., has not yet been garbage-collected. (This is proved as part of Lemma 6.17.) Consider the case where an operation uses some configuration c_k, but does not use configuration c_{k+1} (or any larger-indexed configuration). In addition, assume that some (possibly concurrent) garbage-collection operation removes c_k. We can ascertain that there is some node i that is accessed by both the read/write operation and the garbage-collection. In order for the tag to be propagated correctly, we must show that i learns about the read/write operation before responding to the garbage-collection, thus ensuring that c_{k+1} learns about the read/write operation. However, if this were not the case, then i would send a response indicating the existence of configuration c_{k+1}, and the read/write operation would use the new configuration. From this we conclude that the tag is successfully propagated.

Claim 2. The second claim, Corollary 6.11, shows that once a tag is known to some configuration c_k that is still active, then it is correctly propagated from one configuration to the next. Here, the key is to show that each garbage-collection discovers the relevant tags during its query phase, and relays them during its propagation phase. In many ways, the main machinery in the proof lies in showing this claim.

Claim 3. The third claim, Lemma 6.14, shows that the tag is propagated to the second operation. Specifically, it identifies a configuration that must be active when the second operation begins, and shows that the second operation retrieves the tag from this configuration during its query phase.

Putting these three claims together, we show in Lemma 6.17 that the tags induce the desired ordering, and hence that operations are atomic.

Overview. The proof is carried out in several stages. First, in Section 6.2, we establish some notational conventions and define some useful history variables. In Sections 6.3 and 6.4, we present some basic invariants and guarantees. Next, in Section 6.5 we show that tags are propagated from one configuration to the next by garbage-collection operations. In Section 6.6 we show how tags are propagated between read/write operations and configurations. In Section 6.7, we show that tags are properly ordered by consider the relationship between two read or write operations. Finally, Section 6.8 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 requests are well-formed (as per Definition 4.1). We call these *good* executions. In particular, an "invariant" in this section is a statement that is true of all states that are reachable in good executions of $\mathcal{S}$.

6.2 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 $\mathit{pnum\text{-}local}_i$ just after the read, write, or gc event of the operation occurs. We introduce the following history variables:

- $\mathit{in\text{-}transit}$, a set of messages, initially $\emptyset$.
  A message is added to the set when it is sent by any $\mathit{Reader\text{-}Writer}_i$ to any $\mathit{Reader\text{-}Writer}_j$. No message is ever removed from this set.
- For every $k \in \mathbb{N}$:
  - $c(k) \in C$, initially undefined.
    This is set when the first $\mathsf{new\text{-}config}(c, k)_i$ occurs, for some c and i. It is set to the c that appears as the first argument of this action. Since, by assumption, the Recon service guarantees agreement, we know that the same configuration $c(k)$ is installed at every participant that is notified of configuration k.
- For every operation π:
  - $\mathit{tag}(\pi) \in T$, 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:

- $query\text{-}cmap(\pi)$, a configuration map, initially undefined.
  This is set in the query-fix step of π, to the value of $op.cmap$ in the pre-state.
- $R(\pi, k)$, for $k \in \mathbb{N}$, a subset of I, initially undefined.
  This is set in the query-fix step of 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.
- $prop\text{-}cmap(\pi)$, a configuration map, initially undefined.
  This is set in the prop-fix step of π, to the value of $op.cmap$ in the pre-state.
- $W(\pi, k)$, for $k \in \mathbb{N}$, a subset of I, initially undefined.
  This is set in the prop-fix step of 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:
  - $removal\text{-}set(\gamma)$, 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.
  - $target(\gamma)$, a configuration, initially undefined.
    This is set in the gc(k) step of γ to k.
  - $R(\gamma, \ell)$, 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.
  - $W_1(\gamma, \ell)$, 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.
  - $W_2(\gamma)$, 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):

- For every read or write operation π:
  - 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.
  - 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.

6.3 Configuration map invariants

In this section, we give invariants showing that the configuration maps remain usable. This implies that there is always some active configuration available. We begin with a lemma saying that the $update$ operation on configuration maps preserves usability:

Lemma 6.2 *If $cm, cm' \in Usable$ then $update(cm, cm') \in Usable$.*

Proof. Immediate, by the definition of $update$. □

We next show that configuration maps remain usable:

Invariant 1 *Let cm be a configuration map that appears as one of the following: (1) The cm component of some message in $in\text{-}transit$; (2) $cmap_i$ for any $i \in I$; (3) $op.cmap_i$ for some $i \in I$ for which $op.phase \neq idle$; (4) $query\text{-}cmap(\pi)$ or $prop\text{-}cmap(\pi)$ for any operation π; (5) $gc.cmap_i$ 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 configuration map to become unusable. □

6.4 Phase guarantees

In this section, we present results on the behavior of query and propagation phases for both read/write operations and garbage collection operations. We give four lemmas, describing the information flow during each phase. Specifically, each of these lemmas asserts that when some node i completes a phase, then for every node j in some set of quorums, there are some messages m and m' that convey information from i to j and back from j to i, that is:

- m is sent from i to j after the phase begins.
- m' is sent from j to i after j receives m.
- m' is received by i before the end of the phase.
- In the case of a query phase, the tag of i at the end of the phase is at least as large as the tag of j when message m' is sent. In the case of a propagate phase, then the tag of j at the end of the phase is at least as large as the tag of the read/write/garbage-collection operation.

Additionally, these lemmas make claims about the $cmap$ associated with the operation or garbage collection. Essentially, these lemmas argue that the handhshake protocol for determing a "recent" message (based on phase numbers) works correctly.

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 node to itself is treated uniformly with communication between two different nodes.

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

Lemma 6.3 *Suppose that a* $\mathsf{query\text{-}fix}_i$ *event for a read or write operation* π *occurs in an execution* α. *Let* $k, k' \in \mathbb{N}$. *Suppose* $query\text{-}cmap(\pi)(k) \in C$ *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* $\mathsf{query\text{-}phase\text{-}start}(\pi)$ *event.*
2. *m' is sent after j receives m.*
3. *m' is received before the* $\mathsf{query\text{-}fix}$ *event of* π.
4. *If t is the value of tag_j in any state before j sends m', then:*
   (a) *$tag(\pi) \geq t$.*
   (b) *If π is a write operation then $tag(\pi) > t$.*
5. *If $cmap(\ell)_j \neq \bot$ for all $\ell \leq k'$ in any state before j sends m', then $query\text{-}cmap(\pi)(\ell) \in C$ 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 $tag(\pi) \geq t$, or, in the case of a write operation, $tag(\pi) > t$.

Next, assume that $cmap(\ell)_j \neq \bot$ for all $\ell \leq k'$ prior to j sending message m'. Since i receives m' after the $\mathsf{query\text{-}phase\text{-}start}(\pi)$ event, we can conclude that after receiving m', $op.cmap(\ell)_i \neq \bot$ for all $\ell \leq k'$. Since $op.cmap_i$ is $Usable$, as per Invariant 1, we conclude that $op.cmap_i(\ell) \in C$ for some $\ell \geq k'$, implying the desired claim. □

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

Lemma 6.4 *Suppose that a* $\mathsf{prop\text{-}fix}_i$ *event for a read operation or write operation* π *occurs in an execution* α. *Suppose* $prop\text{-}cmap(\pi)(k) \in C$ *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* $\mathsf{prop\text{-}phase\text{-}start}(\pi)$ *event.*
2. *m' is sent after j receives m.*
3. *m' is received before the* $\mathsf{prop\text{-}fix}$ *event of* π.
4. *In any state after j receives m, $tag_j \geq tag(\pi)$.*
5. *If $cmap(\ell)_j \neq \bot$ for all $\ell \leq k'$ in any state before j sends m', then $prop\text{-}cmap(\pi)(\ell) \in C$ 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 $\geq tag(\pi)$, ensuring that after j receives message m, $tag_j \leq tag(\pi)$. The final conclusion is identical to Lemma 6.3, with respect to the $prop\text{-}cmap(\pi)$ rather than the $query\text{-}cmap(\pi)$. □

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 6.5 *Suppose that a* $\mathsf{gc\text{-}query\text{-}fix}(k)_i$ *event for garbage-collection operation* γ *occurs in an execution* α *and* $k' \in removal\text{-}set(\gamma)$. *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 $\mathsf{gc}(k)_i$ event of γ.*
2. *m' is sent after j receives m.*
3. *m' is received before the $\mathsf{gc\text{-}query\text{-}fix}(k)_i$ event of γ.*
4. *If t is the value of tag_j in any state before j sends m', then $tag(\gamma) \geq t$.*
5. *In any state after j receives m, $cmap(\ell)_j \neq \bot$ 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 $tag(\gamma) \geq t$.

The final claim holds since, when the $\mathsf{gc}(k)_i$ event occurs, we know that $cmap(\ell)_i \neq \bot$ 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 6.6 *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, $tag_j \geq tag(\gamma)$.*

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 $\geq tag(\gamma)$, ensuring that after j receives message m, $tag_j \geq tag(\gamma)$. □

6.5 Garbage collection

This section establishes lemmas describing information flow between garbage-collection operations. The key result in this section is Lemma 6.9, 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, however, 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 6.7 *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 configuration map 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 $op.cmap_i$, for any $i \in I$.*
4. *The $gc.cmap_i$, 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 configuration map $cm(k-1) = \pm$ 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 6.8 *Assume that a* $\mathsf{gc}(k)_i$ *event occurs in an execution* α*, associated with garbage collection* γ*. Let* $k' = \min\{\textit{removal-set}(\gamma)\}$*, 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, $\textit{cmap}(k'-1)_i = \pm$ and $\textit{cmap}(k')_i \neq \pm$. Lemma 6.7 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 6.9 *If a* gc_i *event for garbage-collection operation* γ *occurs in an execution* α *such that* $k \in \textit{removal-set}(\gamma)$*, 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 $k \in \textit{removal-set}(\gamma_k)$; 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 $\ell - 1 \in \textit{removal-set}(\gamma_\ell)$, then let $\gamma_{\ell-1} = \gamma_\ell$, and all the necessary properties still hold.

Otherwise, $\ell - 1 \notin \textit{removal-set}(\gamma_\ell)$ and $\ell \in \textit{removal-set}(\gamma_\ell)$, which implies that $\ell = \min\{\textit{removal-set}(\gamma_\ell)\}$. By Corollary 6.8, 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) $\textit{target}(\gamma_{\ell-1}) = \min\{k' : k' \in \textit{removal-set}(\gamma_\ell)\}$, i.e., $\textit{target}(\gamma_{\ell-1}) = \ell$.

Since $\textit{removal-set}(\gamma_{\ell-1}) \neq \emptyset$, by definition and as a result of the precondition of a gc event, this implies that $\ell - 1 \in \textit{removal-set}(\gamma_{\ell-1})$, 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, $\textit{tag}(\gamma_s)$ is nondecreasing in s.

Lemma 6.10 *Let* $\gamma_\ell, \ldots, \gamma_k$ *be an* (ℓ, k)*-gc-sequence of garbage-collection operations. Then* $\forall\ s : \ell \leq s < k,\ \textit{tag}(\gamma_s) \leq \textit{tag}(\gamma_{s+1})$.

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 $k_2 \in \textit{removal-set}(\gamma_{s+1})$. 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 6.6, and the monotonicity of $\textit{tag}_j$, after the gc-prop-fix event of γ_s we know that $\textit{tag}_j \geq \textit{tag}(\gamma_s)$. Then by Lemma 6.5 $\textit{tag}(\gamma_{s+1}) \geq \textit{tag}_j$. Therefore $\textit{tag}(\gamma_s) \leq \textit{tag}(\gamma_{s+1})$. □

Corollary 6.11 *Let* $\gamma_\ell, \ldots, \gamma_k$ *be an* (ℓ, k)*-gc-sequence of garbage-collection operations. Then* $\forall\ s, s' : \ell \leq s \leq s' \leq k$, $\textit{tag}(\gamma_s) \leq \textit{tag}(\gamma_{s'})$

Proof. This follows immediately from Lemma 6.10 by induction. □

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

Now we describe the relationship between a garbage collection operation and a later read or write operation.. The key result in this section, Lemma 6.14, shows that if a garbage-collection operation completes prior to some read or write operation, then the tag of the garbage collection is less then or equal to the tag of the read or write operation.

The first lemma considers a read/write operation that does *not* use some configuration with index $k-1$; in this case, there must be some garbage collection operation that removes configuration $k - 1$ prior to the operation. More specifically, 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 6.12 *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 $\mathit{query\text{-}cmap}(\pi)(k) \in C$. Assume $k > 0$. Then there exists a garbage-collection operation γ such that $k = \mathit{target}(\gamma)$, and the* gc-prop-fix *event of γ precedes the* query-phase-start(π) *event.*

Proof. This follows immediately from (the contrapositive of) Lemma 6.7. □

Second, in this case where some read/write operation does not use configuration $k-1$, then there is some configuration with index $\geq k$ that is used by the read/write operation. More specifically, if a garbage collection that removes $k-1$ completes before the query-phase-start event of a read or write operation, then some configuration with index $\geq k$ must be included in the $\mathit{query\text{-}cmap}$ of the later read or write operation. (If this were not the case, then the read or write operation would have no extant configurations available to it.)

Lemma 6.13 *Let γ be a garbage-collection operation such that $k-1 \in \mathit{removal\text{-}set}(\gamma)$. 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 $\mathit{query\text{-}cmap}(\pi)(\ell) \in C$ for some $\ell \geq k$.*

Proof. Suppose, for the sake of contradiction, that the conclusion does not hold, i.e., that $\mathit{query\text{-}cmap}(\pi)(\ell) \notin C$ for all $\ell \geq k$. Fix $k' = \max(\{\ell' : \mathit{query\text{-}cmap}(\pi)(\ell') \in C\})$. Then $k' < k$.

Let $\gamma_0, \ldots, \gamma_{k-1}$ be a $(0, k-1)$-gc-sequence of garbage-collection operations whose existence is asserted by Lemma 6.9, where $\gamma_{k-1} = \gamma$. Then, $k' \in \mathit{removal\text{-}set}(\gamma_{k'})$, and the gc-prop-fix event of $\gamma_{k'}$ precedes (or is equal to) the gc-prop-fix event of γ in α, and hence precedes the query-phase-start(π) event in α.

Since $k' \in \mathit{removal\text{-}set}(\gamma_{k'})$, write-quorum $W_1(\gamma_{k'}, k')$ is defined. Since $\mathit{query\text{-}cmap}(k') \in C$, the read-quorum $R(\pi, k')$ is defined. Choose $j \in W_1(\gamma_{k'}, k') \cap R(\pi, k')$. Assume that $k_t = \mathit{target}(\gamma_{k'})$. Notice that $k' < k_t$. Then Lemma 6.5 and monotonicity of cmap imply that in the state just prior to the gc-query-fix event of $\gamma_{k'}$, $\mathit{cmap}(\ell)_j \neq \perp$ for all $\ell \leq k_t$. Then Lemma 6.3 implies that $\mathit{query\text{-}cmap}(\pi)(\ell) \in C$ for some $\ell \geq k_t$. But this contradicts the choice of k'. □

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

Lemma 6.14 *Let γ be a garbage-collection operation, and assume that $k = \mathit{target}(\gamma)$. 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 $\mathit{query\text{-}cmap}(\pi)(k) \in C$. Then:*

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

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

6.7 Behavior of sequential reads and writes

We focus on the case where two read/write operations execute sequentially (i.e., the first completes before the second begins), and we prove some relationships between their configuration maps and and tags. The first lemma says that for two such read/write operations, the second operation uses configurations with indices at least as large as those used by the first operation. More specifically, it shows that 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. Thus, we cannot have a situation in which the second operation's query phase executes using only configurations with indices that are strictly less than any used in the first operation's propagation phase.

Lemma 6.15 *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 $\min(\{\ell : \textit{prop-cmap}(\pi_1)(\ell) \in C\}) > k$, where k is defined to be $\max(\{\ell : \textit{query-cmap}(\pi_2)(\ell) \in C\})$. Then in particular, $\textit{prop-cmap}(\pi_1)(k) \notin C$. The form of $\textit{prop-cmap}(\pi_1)$, as expressed in Invariant 1, implies that $\textit{prop-cmap}(\pi_1)(k) = \pm$.

This implies that some gc-prop-fix event for some garbage-collection operation γ such that $k \in \textit{removal-set}(\gamma)$ occurs prior to the prop-fix event of π_1, and hence prior to the query-phase-start(π_2) event signalling the beginning of π_2. Lemma 6.13 then implies that $\textit{query-cmap}(\pi_2)(\ell) \in C$ for some $\ell \geq k+1$. But this contradicts the choice of k. □

The next lemma describes propagation of $\textit{tag}$ information in the case where the propagation phase of the first operation and the query phase of the second operation share a configuration.

Lemma 6.16 *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) $\textit{prop-cmap}(\pi_1)(k)$ and $\textit{query-cmap}(\pi_2)(k)$ are both in C. Then:*

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

Proof. The hypotheses imply that $\textit{prop-cmap}(\pi_1)(k) = \textit{query-cmap}(\pi_2)(k) = c(k)$. Then $W(\pi_1, k)$ and $R(\pi_2, k)$ are both defined in 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 6.4 implies that, in any state after the prop-fix event of π_1, $\textit{tag}_j \geq \textit{tag}(\pi_1)$. Since the prop-fix event of π_1 precedes the query-phase-start(π_2) event, we have that $t \geq \textit{tag}(\pi_1)$, where t is defined to be the value of $\textit{tag}_j$ just before the query-phase-start(π_2) event. Then Lemma 6.3 implies that $\textit{tag}(\pi_2) \geq t$, and if π_2 is a write operation, then $\textit{tag}(\pi_2) > t$. 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 6.17 *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. *$\textit{tag}(\pi_1) \leq \textit{tag}(\pi_2)$.*
2. *If π_2 is a write then $\textit{tag}(\pi_1) < \textit{tag}(\pi_2)$.*

Proof. Let i_1 and i_2 be the indices of the processes that run operations π_1 and π_2, respectively. Let $cm_1 = \textit{prop-cmap}(\pi_1)$ and $cm_2 = \textit{query-cmap}(\pi_2)$. If there exists k such that $cm_1(k) \in C$ and $cm_2(k) \in C$, then Lemma 6.16 implies the conclusions of the lemma. So from now on, we assume that no such k exists.

Lemma 6.15 implies that $\min(\{\ell : cm_1(\ell) \in C\}) \leq \max(\{\ell : cm_2(\ell) \in C\})$. Invariant 1 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 6.12 implies that there exists a garbage-collection operation that we will call γ_{s_2-1} such that $s_2 = \textit{target}(\gamma_{s_2-1})$, and the gc-prop-fix of γ_{s_2-1} precedes the query-phase-start(π_2) event. Then by Lemma 6.14, $\textit{tag}(\gamma_{s_2-1}) \leq \textit{tag}(\pi_2)$, and if π_2 is a write operation then $\textit{tag}(\gamma_{s_2-1}) < \textit{tag}(\pi_2)$.

Next we will demonstrate a chain of garbage-collection operations with non-decreasing tags. Lemma 6.9, 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 $s_1 \in \textit{removal-set}(\gamma_{s_1})$. Then Corollary 6.11 implies that $\textit{tag}(\gamma_{s_1}) \leq \textit{tag}(\gamma_{s_2-1})$.

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 6.4 for the propagation phase of γ_{s_1}. Let message m_2 be the message from the process running γ_{s_1} to j, and let message m_2' be the message from j to the process running γ_{s_1}, as must exist according to Lemma 6.5 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 $s_t = \textit{target}(\gamma_{s_1})$. Notice that $s_t > s_1$, since $s_1 \in \textit{removal-set}(\gamma_{s_1})$.

Lemma 6.5 implies that in any state after j receives m_2, before j sends m'_2, $cmap(k)_j \neq \perp$ 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, $cmap(k)_j \neq \perp$ for all $k \leq s_t$. Then Lemma 6.4 implies that $prop\text{-}cmap(\pi_1)(\ell) \in C$ 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 6.4 implies that, at the time j sends m'_2, $tag(\pi_1) \leq tag_j$. Then Lemma 6.5 implies that $tag(\pi_1) \leq tag(\gamma_{s_1})$. From above, we know that $tag(\gamma_{s_1}) \leq tag(\gamma_{s_2-1})$, and $tag(\gamma_{s_2-1}) \leq tag(\pi_2)$, and if π_2 is a write operation then $tag(\gamma_{s_2-1}) < tag(\pi_2)$. Combining the various inequalities then yields both conclusions. □

6.8 Atomicity

Let β be a well-formed trace of $\mathcal{S}$ that satisfies Definition 4.1, and assume that all read and write operations complete in β. Consider any particular (good) execution α of $\mathcal{S}$ whose trace is β. 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 no larger than t and before all writes with tags larger than t.

Lemma 6.18 *The ordering $\prec$ is well-defined.*

Proof. We need to show that no two write operations are assigned the same tag. This is clearly true for two writes that are initiated at different nodes, because the low-order tiebreaker identifiers are different. For two writes at the same node, Lemma 6.17 implies that the tag of the second is greater than the tag of the first. □

Lemma 6.19 *$\prec$ satisfies the four conditions in the definition of atomicity in Definition 4.3.*

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:

1. π_2 is a write operation.
   Since π_1 completes before π_2 starts, Lemma 6.17 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 6.17 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 follows from the fact that each write has a unique tag associated with it; as a result, all write operations are ordered. (By the definition of the partial ordering, each read operation is ordered with respect to each write operation.)

Property 4 can be observed as follows. For each read operation π_R, there is clearly some write operation π_W where $tag(\pi_R) = tag(\pi_W)$. It is easy to see that the read operation π_R returns the value that was written by π_W, as tags and values are propagated together by gossip messages. Moreover, by the manner in which the partial order $\prec$ is defined, π_W is the "largest" write operation that precedes π_R, and hence Property 4 is satisfied. □

Putting everything together, we conclude with Theorem 6.1.

Theorem 6.1 RAMBO *implements a reconfigurable atomic memory, as in Definition 4.4. That is, let β be a trace of the system $\mathcal{S}$. If requests in β are well-formed (as in Definition 4.1), then responses are well-formed (as in Definition 4.2) and operations in β are atomic (as in Definition 4.3). That is, $\mathcal{S}$ is a Reconfigurable Atomic Memory (Definition 4.4).*

Proof. Let β be a trace of $\mathcal{S}$ that satisfies Definitions 4.1. We argue that β satisfies Definitions 4.2 and 4.3. The proof that β satisfies the RAMBO well-formedness guarantees is straightforward from the code. To show that β satisfies the atomicity condition, 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 6.19 says that $\prec$ satisfies the four conditions in the definition of atomicity. Thus, β satisfies the atomicity condition, as needed. □

7 Implementation of the Reconfiguration Service

In this section, we describe a distributed algorithm that implements the $Recon$ service. The reconfiguration service is built using a collection of global consensus services $Cons(k, c)$, one for each $k \geq 1$ and for each $c \in C$. The protocol presented in this section is responsible for coordinating the various consensus instances.

First, in Section 7.1, we describe the consensus service $Cons(k, c)$, which can be implemented using the Paxos consensus algorithm [38]. We then in Section 7.2 describe the $Recon$ automata that, together with the consensus components, implement the reconfiguration service.

7.1 Consensus service

In this subsection, we specify the behavior of the consensus service $Cons(k, c)$ for a fixed $k \geq 1$ and $c \in C$. The external signature of $Cons(k, c)$ is given in Figure 8. The goal of the consensus service is to reach agreement among members of configuration c. The protocol is initiated when members of configuration c submit proposals (via $init$ inputs) for the next configuration. Eventually, the consensus service outputs one of the proposals as its decision (via $decide$ outputs).

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

Figure 8: $Cons(k, c)$: External signature

We begin by stating the environmental well-formedness assumptions:

Definition 7.1 (Consensus Input Well-Formedness) For every integer k and configuration c, for every $i \in members(c)$:

1. *Failures:* After a fail_i event, there are no further $\mathsf{init}(*)_{k,c,i}$ events.
2. *Initialization:* There is at most one $\mathsf{init}(*)_{k,c,i}$ event.

In every execution satisfying the well-formedness properties, the consensus service guarantees the following safety properties:

Definition 7.2 (Consensus Safety) For every integer k and configuration c:

1. *Well-formedness:* For every $i \in members(c)$:
   - After a fail_i event, there are no further $\mathsf{decide}(*)_{k,c,i}$ events.
   - At most one $\mathsf{decide}(*)_{k,c,i}$ event occurs.
   - If a $\mathsf{decide}(*)_{k,c,i}$ event occurs, then it is preceded by an $\mathsf{init}(*)_{k,c,i}$ event.
2. *Agreement:* If $\mathsf{decide}(v)_{k,c,i}$ and $\mathsf{decide}(v')_{k,c,i'}$ events occur, then $v = v'$.
3. *Validity:* If a $\mathsf{decide}(v)_{k,c,i}$ event occurs, then it is preceded by an $\mathsf{init}(v)_{k,c,j}$.

Each consensus service $Cons(k, c)$ can be implemented using a separate instance of the Paxos algorithm [38]. This satisfies the safety guarantees described above:

Theorem 7.3 *If for some k and c, β is a trace of $Paxos$ that satisfies Definition 7.1, then β also satisfies the well-formedness, agreement, and validity guarantees (Definition 7.2).*

7.2 Recon automata

The $Recon$ automata coordinate the collection of consensus services, translating a reconfiguration request into an input for the appropriate consensus service, and translating decisions into an appropriate (ordered) sequence of configurations. The signature and state of $Recon_i$ appear in Figures 9, and the transitions in Figure 10.

We now briefly review the interface of the reconfiguration service, which was specified in Section 4.2. Recall that the the service is activated by the joining protocol, which performs a join input. When the service is activated, it responds with a $\mathsf{join\text{-}ack}$ response. The reconfiguration service accepts recon requests, and responds with $\mathsf{recon\text{-}ack}$ responses (irregardless of whether the reconfiguration has succeeded or not). When a new configuration is chosen, the service outputs $\mathsf{report}(c)$, indicating that configuration c has been agreed upon. In response to $\mathsf{request\text{-}config}(k)$ requests, it also specifies the ordering of configurations via $\mathsf{new\text{-}config}(c, k)$ outputs, which specify that c is the k^{th} configuration. The fail input indicates that the node on which the automaton is executing has failed.

Signature:

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

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

State:

$\mathit{status} \in \{\mathit{idle}, \mathit{active}, \mathit{failed}\}$, initially idle.
$\mathit{rec\text{-}cmap} : \mathbb{N} \to C_{\pm}$, a configuration map, initially:
$\mathit{rec\text{-}cmap}(0) = c_0$
$\mathit{rec\text{-}cmap}(k) = \bot$ for all $k \neq 0$.
$\mathit{did\text{-}init} \subseteq \mathbb{N}^+$, initially $\emptyset$
$\mathit{did\text{-}new\text{-}config} \subseteq \mathbb{N}^+$, initially $\emptyset$

$\mathit{cons\text{-}data} \in (\mathbb{N}^+ \to (C \times C))$: initially $\bot$ everywhere
$\mathit{rec\text{-}status} \in \{\mathit{idle}\} \cup (\mathit{active} \times \mathbb{N}^+)$, initially idle
$\mathit{reported} \subseteq C$, initially $\emptyset$

Figure 9: Recon_i: Signature and state

The Recon automaton at node i also has some additional interface components for communicating with the consensus services $\mathit{Cons}(k, c)$ and the communication channels. For each integer k and configuration c, it has an output $\mathsf{init}_{k,i,c}$ to initialize $\mathit{Cons}(k, c)$, and an input $\mathsf{decide}_{k,i,c}$ for receiving decisions from the consensus service. It sends and receives messages from node j via $\mathsf{send}_{i,j}$ and $\mathsf{recv}_{j,i}$ actions. Notice that unlike the $\mathit{Reader\text{-}Writer}$ automata, the Recon automata do not keep track of all the active participants in the system; thus messages are sent and received only among members of known configurations.

In terms of state, each Recon automaton maintains its current status, either idle, active, or failed, and also the $\mathit{rec\text{-}status}$, which indicates whether a reconfiguration request has been submitted, and if so, with respect to which configuration index. It also maintains a configuration map $\mathit{rec\text{-}cmap}$ which tracks the configuration assigned to each index k. The set $\mathit{did\text{-}init}$ keeps track of the set of integers k such that the automaton has submitted an init request to $\mathit{Cons}(k, \cdot)$. The set $\mathit{did\text{-}config}$ keeps track of the set of integers k such that it has produced an output $\mathit{new\text{-}config}(k, \cdot)$. The $\mathit{cons\text{-}data}$ map stores any ongoing reconfiguration request: if $\mathit{cons\text{-}data}(k) = \langle c, c'\rangle$, that indicates that a reconfiguration has been requested from configuration c, the k^{th} configuration, to configuration c', which will become the $k + 1^{st}$ configuration. The set $\mathit{reported}$ indicates which configuration have been produced via the report output.

The operation of the automaton is straightforward. When it receives a $\mathit{recon}(c, c')$ request, it sets up the reconfiguration, first finding the appropriate index k of configuration c and updating the $\mathit{cons\text{-}data}$ map. This then causes an $\mathit{init}(c')_{k,c,i}$, initiating the $\mathit{Cons}(k, c)$ consensus service with a proposal for new configuration c'. (It only proceeds with this initialization after all previous configurations have been output, and only if there has been no prior initialization for $\mathit{Cons}(k, \cdot)$.) When the consensus instance completes, the Recon automaton is notified via a $\mathsf{decide}(c'')_{k,c,i}$ as to the new configuration. This new configuration is then stored in the configuration map $\mathit{rec\text{-}cmap}$, and the reconfiguration request completes with a recon-ack. The new configuration is first installed via a new-config (once it is requested), and then reported to the client via a report.

Lastly, notice that the $\mathit{rec\text{-}cmap}$ and the $\mathit{cons\text{-}data}$ are both continuously sent to other nodes via background gossip. This gossip ensures that if some configuration c is chosen to be the k^{th} configuration, then every node in configuration c eventually learns about this; and the gossip ensures that if some node proposes a $\mathsf{recon}(c, c')$, then eventually every member of configuration c learns about this reconfiguration and issues an init to begin the appropriate consensus instance.

It is easy to see that the reconfiguration service meets the required specification:

Theorem 7.4 *The Recon automata implement a reconfiguration service (as per Definition 4.8).*

8 Performance and Fault-Tolerance Hypotheses

In this and the following two sections, Sections 9 and 10, we present our conditional performance results—an analysis of the latency of RAMBO operations under various assumptions about timing, failures, and the patterns of requests. We present the results in two groups: Section 9 contains results for executions in which "normal timing behavior" is observed throughout the execution; Section 10 contains results for executions that "stabilize" so that "normal timing behavior" is observed from some

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

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

Input $\mathsf{request\text{-}config}(k)_i$
Effect:
    if $status \notin \{idle, failed\}$ then
        $ready\text{-}new\text{-}config \leftarrow ready\text{-}new\text{-}config \cup \{k\}$

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

Output $\mathsf{report}(c)_i$
Precondition:
    $status = active$
    $c \notin reported$
    $c \in did\text{-}new\text{-}config$
Effect:
    $reported \leftarrow reported \cup \{c\}$

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

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

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

Output $\mathsf{recon\text{-}ack}()_i$
Precondition:
    $status = active$
    $rec\text{-}status = \langle active, k\rangle$
    $rec\text{-}cmap(k) \neq \bot$
Effect:
    $rec\text{-}status \leftarrow idle$
    $outcome \leftarrow \bot$

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

Input $\mathsf{decide}(c')_{k,c,i}$
Effect:
    if $status = active$ then
        $rec\text{-}cmap(k) \leftarrow c'$


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

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

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

Figure 10: $Recon_i$ transitions.

point onward. In this section, we define what we mean by "normal timing behavior," presenting 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 $\mathit{Reader\text{-}Writer}_i$ and Joiner_i for all i, Recon_{impl} (which consists of Recon_i for all i and $\mathit{Cons}(k, c)$ for all k and c), and channels between all i and j. Since we are dealing here with timing, we "convert" all these automata to general timed automata as defined in [43] by allowing arbitrary amounts of time to pass in any state, without changing the state.

This section is divided into the following parts. First, in Section 8.1, we restrict the nondeterminism in RAMBO, ensuring that locally-controlled events occur in a timely fashion. Next, in Section 8.2 we specify the behavior or the network, assuming that (eventually), messages are delivered in some bounded time. Finally, in Sections 8.3–8.7, we present assumptions on the viability of installed configurations, the rate at which nodes join, and the rate of reconfiguration.

8.1 Restricting nondeterminism

RAMBO in its full generality is a highly nondeterministic algorithm. For example, it allows sending of gossip messages at arbitrary times. 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. We also assume that locally controlled events unrelated to sending messages occur just once, as soon as they are enabled. We now proceed in more detail.

We begin by fixing a constant $d > 0$, which we refer to as the *normal message delay*. We assume a restricted version of RAMBO in which each $\mathit{Reader\text{-}Writer}_i$, 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 $\mathit{status}_i = \mathit{joining}$ sends join messages to everyone in its hints_i set every time d, according to its local clock. Each $\mathit{Reader\text{-}Writer}_i$ sends messages to everyone in its world_i set every time d, according to its local clock.
- *Important Joiner messages:* Whenever the Joiner_i service at node i receives a $\mathsf{join}(\mathit{rambo}, J)_i$ request, it immediately sends a join request to node j, without any time passing on its local clock, for every $j \in J$.
- *Important Reader-Writer messages:* Each $\mathit{Reader\text{-}Writer}_i$ sends a message immediately to node j, without any time passing on its clock, in each of the following situations:
  - Just after a $\mathsf{recv}(\mathit{join})_{j,i}$ event occurs, if $\mathit{status}_i = \mathit{active}$.
  - Just after a $\mathsf{recv}(*, *, *, *, \mathit{snder\text{-}phase}, *)_{j,i}$ event occurs, if $\mathit{snder\text{-}phase} > \mathit{pnum\text{-}vector}(j)_i$ and $\mathit{status}_i = \mathit{active}$.
  - Just after a $\mathsf{new\text{-}config}(c, k)_i$ event occurs if $\mathit{status}_i = \mathit{active}$ and $j \in \mathit{world}_i$.
  - Just after a read_i, write_i, or $\mathsf{query\text{-}fix}_i$ event, or a recv event that resets $\mathit{op.acc}$ to $\emptyset$, if $j \in \mathit{members}(c)$, for some c that appears in the new $\mathit{op.cmap}_i$.
  - Just after a $\mathsf{gc}(k)_i$ event occurs, if $j \in \mathit{members}(\mathit{cmap}(k)_i)$.
  - Just after a $\mathsf{gc\text{-}query\text{-}fix}(k)_i$ event occurs, if $j \in \mathit{members}(\mathit{cmap}(k)_i)$.
- *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 $j \in \mathit{members}(c) - \{i\}$.
  - The message is of the form $(\mathit{init}, c, c', k)$, and an $\mathsf{init}(c')_{k,c,i}$ event has just occurred, for $j \in \mathit{members}(c) - \{i\}$.
- *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: adding such restrictions would unnecessarily complicate the pseudocode (for example, necessitating the use of TIOA [33] to capture timing behavior).

8.2 Normal timing behavior

We now define "normal timing behavior," restricting the timing anomalies observed in an execution, specifically, the reliability of the clocks and the latency of message delivery. We define executions that satisfy this normal timing behavior from some point onwards. An execution that always satisfies normal timing behavior is viewed as a special case. Throughout this section we use the following notation: given an execution prefix α', we define $\ell time(\alpha')$ to be the time of the last event in α'.

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. We assume that after α', good behavior resumes.

Definition 8.1 We say that α is an α'*-normal* execution if the following assumptions hold:

1. *Initial time:* A join-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 in the system progress at exactly the rate of real time, after the prefix α'. (Notice that this does not imply that the clocks are synchronized.)

   Recall from Section 8.1 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 all locally-controlled events occur subject to appropriate timing constraints.
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 if it is delivered, then it is delivered no later than time $\max(t, \ell time(\alpha')) + d$.

8.3 Join-Connectivity

In the remainder of this section, we present several 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.

The first hypothesis bounds the time for two participants that join the system to learn about each other. If they join during a period of normal timing behavior, they learn about each other within e time. If the network is unstable, then within e time of the network stabilizing, they learn about each other.

Definition 8.2 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 join-ack$(rambo)_i$ and a join-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.

8.4 Configuration Viability

The next hypothesis, *configuration-viability*, is a reliability property for configurations, specifically for the quorums that make up each configuration. In general in systems that use quorums, operations are guaranteed to terminate only if certain quorums do not fail. (If no quorums remain available, it is impossible for a quorum-based algorithm to make progress.)

Similarly in this paper, in order to guarantee the termination of read operations, write operations, reconfiguration, and garbage collection, we assume that at least some of the available 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.

Intuitively, we say that a configuration is viable if some of its quorums survive until sufficiently long after the next configuration is installed. The definition is parameterized by a variable τ that indicates for how long a configuration is viable after the next configuration is installed. Notice, however, that during intervals when all messages are lost (i.e., during the unstable prefix of an execution), there is no time τ that is long enough. Thus, an additional requirement is that some quorums remain viable until sufficiently long after the network stabilizes, i.e., after α'. (Thus, α' is also a parameter of the configuration-viability definition). In fact, after the network stabilizes, it may take some time for the join protocol to complete; some quorums must remain extant until sufficiently long after the join protocol completes. A third parameter e captures how long it takes the join protocol to complete.

Installation. We now proceed more formally, first defining the point at which a configuration c is installed; configuration-viability specifies how long prior configurations must remain extant after configuration c is installed. 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 $i \in \mathit{members}(c(k-1))$, either a $\mathsf{decide}(c)_{k,c(k-1),i}$ event or a fail_i event occurs in α. That is, configuration c_0 is installed initially (i.e., always), and a later configuration c is installed when every non-failed member of the prior configuration (i.e., configuration $c(k-1)$) performs a $\mathsf{decide}(c(k))$ event.

Viability. We now define what it means for an execution to be *(α', e, τ)-configuration-viable*:

Definition 8.3 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 every configuration c for which a $\mathsf{report}(c)_i$ event occurs for some i, 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. Fix k such that c is the k^{th} configuration (as determined by the sequence of reconfigurations). 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 $\ell time(\alpha_{install}) + \tau$.

Notice, then, that if a configuration is viable, there is at least one read-quorum and at least one write-quorum that survives until either time $e + 2d + \tau$ after the network stabilizes (i.e., after α'), or at least time τ after the next configuration is installed.[3]

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.

Choosing configurations. A natural question that arises is how to choose configurations that will remain viable, and when to initiate reconfigurations so as to ensure viability. In general, this is not a trivial problem and it is well outside the scope of this paper. Even so, it is a critical issue that arises in implementing RAMBO, or *any* dynamic quorum-based system. In part, we believe that configuration-viability is a reasonable assumption (and a problem outside the scope of this paper) as it seems a necessary property of *any* dynamic quorum-based system.

The algorithm responsible for initiating reconfiguration is responsible for observing when sufficiently many nodes have failed to jeopardize a configuration, and to propose a new configuration sufficiently early that it can be installed prior to the old configuration failing. This task can be divided into three components:

1. First, the reconfigurer must monitor the availability of quorums to determine whether a configuration is at risk. (This monitoring may take the form of a *failure detector* (see [13]) which guesses when other nodes in the system have crashed. Failure detectors can achieve sufficient reliability when the network is stable.)

2. It must also evaluate the *rate* at which failures occur, in order to determine how long the remaining quorums can be expected to survive. (This estimate of how long a given node is likely to survive can also enable the selection of new configurations with maximum viability.)

3. Finally, it must estimate the network latencies and rate of message loss, in order to gauge how long it will take to install the new configuration: when the network is stable, consensus (and hence reconfiguration) can be quite fast; when the network is unstable, it might be quite slow. (Notice, then, that configuration viability depends on the operation of the $Recon$ service, in that a new configuration must be proposed sufficiently far in advance for $Recon$ to install it prior to the previous configuration failing; a well-implemented $Recon$ service can provide the necessary latency guarantees during periods of network stability.)

Thus, we believe that if intervals of network instability are of bounded length, and if the rate of node failures is bounded, and if the rate of newly joining nodes is sufficient to compensate for failures, then it should be possible to ensure configuration viability. We leave it as an open problem to determine under what precise conditions configuration viability can be achieved, and how to achieve it.

[3] The $e + 2d$ time captures the additional time after the network stabilizes that a quorum must survive. This additional time allows sufficient information to propagate to everyone: within time e after the network stabilizes, each pair of nodes is aware of each other; within a further d time each node sends a gossip message; within a further d time, the gosssip messages are received.

8.5 Recon-Spacing

The next hypothesis states that recon events do not occur too frequently: a $\text{recon}(c,*)$ that initiates a transition from configuration c to a new configuration is only initiated sufficiently long after configuration c was proposed, and after sufficiently many nodes have learned about configuration c. This ensures that successful recon events are not spaced too closely, and that the nodes participating in the recon event are ready to choose a new configuration.

As in the case of configuration viability, the recon-spacing assumption is parameterized by α', the prefix during which timing assumptions may not hold, and e, a constant that indicates how long it takes a new node to join the system.

Definition 8.4 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 $\text{recon}(c,*)_i$ event in α there exists a write-quorum $W \in \mathit{write\text{-}quorums}(c)$ such that for all $j \in W$, $\text{report}(c)_j$ precedes the $\text{recon}(c,*)_i$ event in α.[4]
2. *recon-spacing-2*: For any $\text{recon}(c,*)_i$ event in α after α' the preceding $\text{report(c)}_\text{i}$ event occurs at least time e earlier.

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

8.6 Recon-Readiness

The next hypothesis says that when a configuration c is proposed by some client, every member of configuration c has already joined the system sufficiently long ago.

Definition 8.5 An α'-*normal* execution α satisfies *(α', e)-recon-readiness* if the following property holds: if a $\text{recon}(*,c)_*$ event occurs at time t, then for every $j \in \mathit{members}(c)$:

- A join-ack_j event occurs prior to the recon event.
- If the recon occurs after α', then a join-ack_j event occurs no later than time $t-(e+3d)$.

The delay of $e+3d$ allows time for the join protocol to complete, and for sufficient information to be exchange with other participants. We say simply that α satisfies e-recon-readiness if it satisfies $(\emptyset, e)$-recon-readiness.

8.7 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 8.6 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 join-ack_j event occurs no later than time $t-(e+3d)$.

The delay of $e+3d$ allows time for the join protocol to complete, and for sufficient information to be exchange with other participants. Notice that the gc action is an internally-controlled action, and hence in this case, the hypothesis could be readily enforced via explicit reference to the local clock.

9 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 9.7, shows that every read and write operation completes within $8d$ time, despite concurrent failures and reconfigurations. In fact, each phase (i.e., query or propagation) takes at most time $4d$, as each round-trip takes at most $2d$, and a given phase may require two round-trips (if a new configuration is installed during the phase, requiring a new quorum be contacted). Since each operation takes two phases, the result is an $8d$ bound on each operation. (Notice that if there are no new configurations produced by the *Recon* service during a read or write operation, it is easy to see that the operation completes in time $4d$.)

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. For a timed execution α, we let $\mathit{time}(\pi)$ stand for the real time at which the event π occurs in α.

[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 $\text{report}(c)$ event has occurred.

9.1 Performance Hypotheses

The claims in this section depend on the various hypotheses presented in Section 8. In this section, we list these assumptions, and provide some brief explanation as to how each is used in the analysis. Specifically, we assume that α satisfies: *e-join-connectivity*, *e-recon-readiness*, $11d$*-configuration-viability*, and $13d$*-recon-spacing*. We now proceed in more detail.

First, we fix $e \in \mathbb{R}^{\geq 0}$ such that *e-join-connectivity* holds in α. The hypothesis of join-connectivity states that two nodes are aware of each other soon after they complete the join protocol. Key to the analysis is the fact that information propagates rapidly through the system: whenever a new configuration is installed or an old configuration garbage-collected, this information is soon received by a sufficient subset of the participants (see Lemma 9.1). In order to take advantage of join-connectivity, though, we have to argue that the participants all joined sufficiently long ago.

To this end, we assume that α satisfies *e-recon-readiness*. This implies that if a node is part of configuration c, and if it has received a report about configuration c, then it must have joined sufficiently long ago (Lemma 9.2).

Perhaps the most important performance hypothesis is configuration-viability: we assume that execution α satisfies $11d$*-configuration-viability*. That is, we assume that each configuration is viable for at least $11d$ after the subsequent configuration is installed. The reason we need a configuration to remain viable for $11d$ is to allow sufficient time for a query or propagation phase of an operation to complete.

In order to illustrate this point, assume that some node i is executing a read or write operation. Assume that some new configuration c is installed at time t, and that node i uses some prior configuration c' during either the query or propagation phase. We need to guarantee that quorums of this prior configuration survives for sufficiently long for the phase to complete.

The $11d$ time of viability can be broken down into four components. (1) Time d may elapse from when configuration c is installed to when configuration c is reported. (2) Time $6d$ may elapse during which the old configuration c' is garbage-collection (Lemmas 9.5 and 9.6). (3) Time $2d$ further may elapse while this information is propagated to node i. Notice that any phase that begins after this point will not use any configuration previous to c. Thus we can assume that the query or propagation phase initiated by i begins no later than this point. (4) Time $2d$ may elapse during the query or propagation phase, while i contacts a quorum of c and receives a response. Thus it is sufficient for configuration c' to remain viable for $11d$ from the point at which configuration c is installed.

The last assumption is recon-spacing: we assume that execution α satisfies $13d$*-recon-spacing*. The recon-spacing assumption ensures that a given phase of a read or write operation is interrupted at most once by a new configuration. Recall that whenever a new configuration is discovered, the initiator of the phase must contact a quorum from the new configuration. Thus, each time a new configuration is discovered, the phase may be delivered by a further $2d$ time.

Again, consider the previous example, where some new configuration c is installed at time t, and assume that some phase of a read or write operation uses an old configuration c'. As we just argued, if the query or propagation phase uses old configuration c', then it begins at latest at time $t + 8d$. Thus, the recon-spacing hypothesis ensures that no other configuration is installed until after $t + 12d$, which gives the phase at least time $4d$ to complete. Since there are at most two configurations to contact (i.e., configurations c and c'), we can be certain that the phase will complete by time $t + 12$, and hence $13d$*-recon-spacing* is sufficient.

9.2 Propagation of information

In this section, we show how information is propagated amongst various participants. Specifically, if two nodes both joined sufficiently long ago, then join-connectivity ensures that they exchange information frequently. Thus, if one learns about a new configuration, or learns that a configuration has been garbage collected, then the other learns this as well. One corollary of this is that if a new configuration is installed, soon thereafter every other participant that joined sufficiently long ago learns about the new configuration.

We begin by showing that all participants succeed in exchanging information about configurations, within a short time. If both i and j are "old enough," i.e., 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 9.1 *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, $cmap(k)_i \neq \bot$, then by time $t + 2d$, $cmap(k)_j \neq \bot$.*
2. *If by time t, $cmap(k)_i = \pm$, then by time $t + 2d$, $cmap(k)_j = \pm$.*

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

This next lemma says that a process receiving a report must be "old enough", that is, they have joined at least time e earlier. When combined with Lemma 9.1, this leads to the conclusion that information on newly installed configurations is rapidly propagated.

Lemma 9.2 *Assume that α satisfies e-recon-readiness, $c \in C$, $c \neq c_0$, $i \in members(c)$. 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. Since $c \neq c_0$, we can conclude that the $\mathsf{report}(c)_i$ event is preceded by a $\mathsf{recon}(*, c)_*$ event. The conclusion follows immediately from *e-recon-readiness*. □

The last lemma in this section combines the two previous results to show that if a $\mathsf{report}(c)_i$ event occurs at i and if i does not fail, then another process j that joined sufficiently long ago learns about c soon thereafter.

Lemma 9.3 *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 $c =$ rec-cmap$(k)_i$, and i does not fail by $t + d$.*
2. $\mathsf{join\text{-}ack}(rambo)_j$ *occurs in α by time $t - e$, and j does not fail by time $t + 2d$.*

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

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

Lemma 9.2 implies that $\mathsf{join\text{-}ack}(rambo)_i$ occurs by time $t - e$. By assumption, $\mathsf{join\text{-}ack}(rambo)_j$ occurs by time $t - e$. Also by assumption, i does not fail by time $t + d$, and j does not fail by time $t + 2d$. Furthermore, we claim that, by time t, $cmap(k)_i \neq \perp$ because the $report(c)_i$ occurs at time t; within 0 time, this information gets conveyed to $Reader\text{-}Writer_i$.

Therefore, we may apply Lemma 9.1, to conclude that by time $t + 2d$, $cmap(k)_j \neq \perp$. □

9.3 Garbage collection

The results of this section show that old configurations are garbage-collected just as fast as new configurations are installed. Specifically, if a new configuration c is reported at time t, then by time $t + 6d$ all prior configurations have been garbage collected. In order to show that an old configuration c' is garbage collected within time $6d$, there are three steps. First, in Lemma 9.4, we argue that some member of the old configuration c' survives sufficiently long, i.e., at least time $4d$. Next, in Lemma 9.5, we show that by time $t + 6d$, some set of nodes has garbage collected the old configuration. Finally, in Lemma 9.6, we conclude that in fact every non-failed member of configuration c has removed the old configuration.

We begin by showing that at least one member of the old configuration does not fail for $4d$ time after a new configuration is installed. This follows from the assumption of configuration viability.

Lemma 9.4 *Assume that α satisfies $5d$-configuration-viability, $c \in C$, $k \in \mathbb{N}$, $k \geq 1$, $i \in members(c)$ and $j \in I$, $t \in \mathbb{R}^{\geq 0}$. Suppose:*

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

Then there exists $j \in members(c')$ 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 $\mathsf{report}_{k,c,\ell}$ event at a node $\ell \in members(c')$. 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 next lemma says that a node that has joined sufficiently long ago succeeds in garbage collecting all configurations earlier than c within time $6d$ after the report for configuration c. The argument is structured inductively: assume, for some k, that all configurations prior to $c(k)$ were garbage-collected within time $6d$ of when $c(k)$ was installed; then configuration $c(k)$ is garbage-collected within time $6d$ of configuration $c(k + 1)$ being installed. This lemma relies on recon-spacing to ensure that by the time $c(k + 1)$ is installed, all configurations prior to $c(k)$ have been garbage collected; thus there are no earlier configurations that need to be involved in the garbage-collection of $c(k)$. Lemma 9.4 then ensures that some member of $c(k)$ survives long enough to perform the garbage collection. The remaining details involve ensuring that sufficient information is propagated for the garbage collection to begin and complete.

Lemma 9.5 *Let α be an admissible timed execution satisfying e-recon-readiness, e-join-connectivity, $6d$-recon-spacing and $5d$-configuration-viability, $c \in C$, $k \in \mathbb{N}$, $i \in members(c)$, $j \in I$, $t \in \mathbb{R}^{\geq 0}$. Suppose:*

1. *A $\mathsf{report}(c)_i$ event occurs at time t in α, where $c = \textit{rec-cmap}(k)_i$.*
2. *$\mathsf{join\text{-}ack}(rambo)_j$ 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) $cmap(k-1)_j \neq \bot$ and (b) $cmap(\ell)_j = \pm$ for all $\ell < k - 1$.*
2. *If i does not fail by $t + d$ and j does not fail by time $t + 6d$, then by time $t + 6d$: (a) $cmap(k)_j \neq \bot$ and (b) $cmap(\ell)_j = \pm$ for all $\ell < k$.*

Proof. By induction on k. *Base:* $k = 0$. Part 1 is vacuously true. The clause (a) of Part 2 follows because $cmap(0)_j \neq \bot$ 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), $cmap(0)_j \neq \bot$ 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 9.4 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 $\mathsf{join\text{-}ack}(rambo)_{i''}$ event occurs at least time e prior to the $\mathsf{report}(c')_{i'}$ event. Then by inductive hypothesis, Part 2, by time $time(\mathsf{report}(c')_{i'}) + 6d$, $cmap(k-1)_{i''} \neq \bot$ and $cmap(\ell)_{i''} = \pm$ for all $\ell < k - 1$. By *$6d$-recon-spacing*, $time(\mathsf{recon}(c', c)_{i'}) \geq time(\mathsf{report}(c')_{i'}) + 6d$, and so $t = time(\mathsf{report}(c)_i) \geq time(\mathsf{report}(c')_{i'}) + 6d$. Therefore, by time t, $cmap(k-1)_{i''} \neq \bot$ and $cmap(\ell)_{i''} = \pm$ for all $\ell < k - 1$.

Now we apply Lemma 9.1 to i'' and j, with t in the statement of Lemma 9.1 set to the current t. This allows us to conclude that, by time $t + 2d$, $cmap(k-1)_j \neq \bot$ 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 $t + 6d$. Lemma 9.3 applied to i and j and with t and t' both instantiated as the current t, implies that by time $t + 2d$, $cmap(k)_j \neq \bot$. 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 no later than $t + 2d$. 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 9.6 *Let α be an admissible timed execution satisfying e-recon-readiness, e-join-connectivity, $6d$-recon-spacing and $5d$-configuration-viability, $c \in C$, $k \in \mathbb{N}$, $i \in members(c)$, $j \in I$, $t \in \mathbb{R}^{\geq 0}$. Suppose:*

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

Then:

1. *If $k > 0$ and j does not fail by time $t + 2d$, then by time $t + 2d$, $cmap(k-1)_j \neq \bot$ 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 $t + 6d$, $cmap(k)_j \neq \bot$ 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 $\mathsf{join\text{-}ack}(rambo)_j$ occurs in α by time $t - e$. Then the conclusions follow from Lemma 9.5. □

9.4 Reads and writes

The final theorem bounds the time for read and write operations. There are two key parts to the argument. The first part shows that, due to recon-spacing, each phase of a read or write operation is required to access at most two configurations. This fact follows from Lemma 9.6, which shows that old configurations are removed soon after a new configuration is installed, and thus sufficiently prior to the next reconfiguration. The second part shows that quorums of the relevant configurations survive sufficiently long; this follows from the configuration viability hypothesis.

Theorem 9.7 *Assume that α satisfies e-recon-readiness, e-join-connectivity, $13d$-recon-spacing, and $11d$-configuration-viability, $i \in I$, $t \in \mathbb{R}^+$. Assume that a* read_i *(resp.,* $\mathsf{write}(*)_i$*) event occurs at time t, and* $\mathsf{join\text{-}ack}_i$ *occurs strictly before time $t - (e + 8d)$. Then the corresponding* $\mathsf{read\text{-}ack}_i$ *(resp.,* $\mathsf{write\text{-}ack}(*)_i$*) event occurs by time $t + 8d$.*

Proof. Let $c_0, c_1, c_2, \ldots$ denote the (possibly infinite) sequence of successive configurations decided upon in α. For each $k \geq 0$, if configurations c_k and c_{k+1} exist, let π_k be the first $\mathsf{recon}(c_k, c_{k+1})_*$ event in α, let i_k be the node 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.) Also, for each $k \geq 0$, if configuration c_k exists, choose $s_k \in \mathit{members}(c_k)$ such that: (1) if configuration c_{k+1} exists, then s_k does not fail by time $10d$ after the time of ϕ_{k+1}; (2) otherwise, s_k does not fail in α. The fact that this is possible follows from $11d$-viability (because if there is no configuration c_{k+1}, then configuration c_k is viable forever; otherwise, 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 no more than $4d$—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 $\mathit{time}(\psi) > \mathit{time}(\phi_0) + 8d$, that is, that ψ occurs more than $8d$ time after the $\mathsf{report}(0)_{i_0}$ event: We have that $\mathit{time}(\psi) \geq t$, and $t > \mathit{time}(\mathsf{join\text{-}ack}_i) + 8d$ by assumption. Also, $\mathit{time}(\mathsf{join\text{-}ack}_i) \geq \mathit{time}(\mathsf{join\text{-}ack}_{i_0})$. Furthermore, $\mathit{time}(\mathsf{join\text{-}ack}_{i_0}) \geq \mathit{time}(\phi_0)$, that is, when $\mathsf{join\text{-}ack}_{i_0}$ occurs, $\mathsf{report}(0)_{i_0}$ occurs with no time passage.

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

Next, we claim that by $\mathit{time}(\phi_k)+6d$, $\mathit{cmap}(k)_{s_k} \neq \bot$ and $\mathit{cmap}(\ell)_{s_k} = \pm$ for all $\ell < k$; this follows from Lemma 9.6, Part 2, applied with $i = i_k$ and $j = s_k$, because i_k does not fail before 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, $\mathit{cmap}(k)_i \neq \bot$ and $\mathit{cmap}(\ell)_i = \pm$ for all $\ell < k$: We apply Lemma 9.1 to s_k and i, with t in that lemma set to $\max(\mathit{time}(\phi_k) + 6d, \mathit{time}(\mathsf{join\text{-}ack}_i) + e)$. This yields that, no later than time $\max(\mathit{time}(\phi_k) + 6d, \mathit{time}(\mathsf{join\text{-}ack}_i) + e) + 2d$, $\mathit{cmap}(k)_i \neq \bot$ and $\mathit{cmap}(\ell)_i = \pm$ for all $\ell < k$. Our choice of k implies that $\mathit{time}(\phi_k)+8d < \mathit{time}(\psi)$. Also, by assumption, $\mathit{time}(\mathsf{join\text{-}ack}_i)+e+2d < t$. And $t \leq \mathit{time}(\psi)$. So, $\mathit{time}(\mathsf{join\text{-}ack}_i)+e + 2d < \mathit{time}(\psi)$. Putting these inequalities together, we obtain that $\max(\mathit{time}(\phi_k) + 6d, \mathit{time}(\mathsf{join\text{-}ack}_i) + e) + 2d < \mathit{time}(\psi)$. It follows that, in the pre-state of ψ, $\mathit{cmap}(k)_i \neq \bot$ and $\mathit{cmap}(\ell)_i = \pm$ for all $\ell < k$, as needed.

Now, by choice of k, we know that $\mathit{time}(\psi) \leq \mathit{time}(\phi_{k+1}) + 8d$. The recon-spacing condition implies that $\mathit{time}(\pi_{k+1})$ (the first recon event that requests the creation of the $(k + 2)^{nd}$ configuration) is $> \mathit{time}(\phi_{k+1}) + 12d$. Therefore, for an interval of time of length $> 4d$ after 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 $\mathit{time}(\psi)$, 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 $11d$. □

10 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 10.20, is analogous to Theorem 9.7 in that it shows that every read and write operation completes within $8d$ time, despite concurrent failures and reconfigurations. Unlike Theorem 9.7, however, we do not assume that normal timing behavior always holds; instead, we show that good performance is achieve by read/write operations that occur when 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*,

- $(\alpha',\ e,\ 22d)$-*configuration-viability*,
- $(\alpha',\ 13d)$-*recon-spacing*,
- $(\alpha',\ e)$-*gc-readiness*,

As in Section 9, the join-connectivity assumption ensures that data is rapidly propagated after a node joins the system, and the recon-readiness/gc-readiness assumptions ensure that members of each configuration, especially those initiating garbage-collections, have joined sufficiently long ago. Configuration viability is used to ensure that each configuration remains viable for sufficiently long. The $22d$ time that a configuration should remain viable can be broken down into the following components: (a) $6d$ time from when the new configuration is installed until a garbage-collection operation begins; (b) $8d$ time for the old configuration to be garbage collected: $4d$ for any ongoing garbage-collection operations to complete, and $4d$ for a new garbage-collection operation to remove the old configuration; (c) $5d$ time for the information to propagate appropriately[5]; and (d) $3d$ time for quorums of the old configuration to send responses for any ongoing read/write operations. Finally, recon-spacing ensures that each phase of a read/write operation is interrupted at most once with a new configuration. (The constant $12d$ is chosen for reasons discussed previously in Section 9.)

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. This allows for sufficient time after normal timing behavior resumes for the join protocol to complete, and for nodes to exchange information.

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.

10.1 Overview

There are two key claims that we need to prove, before we can analyze read and write operations: (1) every garbage-collection completes within time $4d$, when good timing behavior holds; and (2) every configuration that is still in use remains viable. In fact, these two key claims are closely related, as per the following argument: (1) Soon after a new configuration is installed, RAMBO can begin to garbage collect the old configurations; we refer to this as the gc-ready event. (2) We know, due to configuration viability, that an old configuration survives for sufficiently long after a new configuration is installed; thus, each configuration survives until sufficiently long after the gc-ready event. (3) Thus, soon after the gc-ready event, a garbage-collection operation begins and contacts the old configuration, which remains viable; soon thereafter, the garbage collection completes and the old configuration is removed. (4) Finally, we conclude that the old configuration has sufficient viability remaining that it will survive until any other ongoing operation completes. A key obstacle in this argument, however, is that older configurations can delay the garbage collection of more recent configurations. Thus, the proof is structured as an inductive argument in which we show that if every previous garbage-collection operation has completed in $4d$, then the current garbage collection also completes in $4d$. Putting the pieces together, this leads to the two main claims described above. We can then conclude that read and write operations terminate within $8d$ using an argument similar to that presented in Theorem 9.7.

Before we can proceed with the main proof, we first, in Section 10.2, prove a sequence of results regarding the propagation of information. This simplifies our later argument in that we can reason easily about what different nodes in the system know. We then define the gc-ready event in Section 10.3, and show that each configuration remains viable for sufficiently long after the associated gc-ready event. Finally, we proceed to the main argument in Section 10.4, where we show that every garbage-collection operation completes in $4d$. We also show as a corollary that every configuration survives for as long as it is in use. We then conclude in Section 10.5, showing that read and write operations terminate in $8d$.

10.2 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 configuration map cm, we say that cm is in the mainstream:

Definition 10.1 Let cm be a configuration map, 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.

We focus on nodes that joined at least time $e + 2d$ ago as this is the set of nodes that have completed the join protocol and had time to exchange information with other participants. In addition, recon-readiness ensures that each member of each configuration has joined at least time $e + 2d$ ago, and hence is aware of every mainstream configuration map.

[5] Notice that a tigheter analysis might be able to reduce the propagation time somewhat.

The main result in this subsection is Theorem 10.6, which shows that once a configuration map is mainstream, then it remains mainstream (despite nodes joining and leaving, and despite ongoing reconfiguration). More specifically, if some configuration map cm is mainstream at some time t_1, then at all times $t_2 \geq t_1 + 2d$, configuration map cm remains mainstream. This result is critical in that it allows for a simplified analysis of how information is propagated: instead of arguing about low-level data propagation, we can simply focus on proving that some configuration map is mainstream. We can then rely on Theorem 10.6 to show that, at some later point in time, information regarding this configuration map is known to the participants.

We begin with a straightforward lemma regarding members of configuration c, showing that (as a result of recon-readiness), they have information on all mainstream configuration maps:

Lemma 10.2 *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, if i is a member of some configuration c, then eventually its $cmap_i$ becomes mainstream:

Lemma 10.3 *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. □

We now proceed to show that once data has become mainstream, it remains mainstream. The main idea is to show that mainstream information is continually propagated forward from members of one configuration to the next. Thus, the first step is to identify the sequence of configurations that are installed and the recon events that proposed them.

Definition 10.4 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 10.6: we assume that a configuration map cm is mainstream after t_1, and show that it remains mainstream after time $t_2 \geq t_1 + 2d$, under the condition that a successful recon event occurs precisely at time t_2. Essentially, the successful recon events that occur after time t_1 pass the information from one configuration to the next. Thus at time t_2, when the successful recon event occurs, we can ensure that some non-failed node is aware of the mainstream configuration map, and within $2d$ time this non-failed node can propagate the information to the other participants.

Lemma 10.5 *Fix times $t_2 \geq t_1 \geq T_{GST} + 2d$. Assume that some configuration map 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 10.2 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. We know from Lemma 10.5 that the configuration map cm is mainstream after some earlier successful recon event (unless there have been no earlier reconfigurations). It remains to show that this information is then propagated to the other participants.

Theorem 10.6 *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 configuration map 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 10.5 shows that cm is mainstream after $t_{rec} + 2d$. Since c was initially proposed at time $t_{rec} < t_{rec} + 2d$, Lemma 10.2 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 10.2 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 10.3 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. □

10.3 Configuration viability

In this subsection, we show that each configuration remains viable for sufficiently long due to configuration viability. We first define an event gc-ready(k), for every $k > 0$; we will show that some garbage collection for configuration $c(k-1)$ begins no later than the gc-ready(k) event. (Garbage-collection operations may, however, occur prior to this event.) We then show in Theorem 10.11 that for every $k \geq 0$, configuration $c(k-1)$ remains viable for at least $16d$ time after the gc-ready(k) event.

We say that a configuration with index k is gc-ready when every smaller configuration can be safely garbage collected, i.e., every smaller configuration has been installed, and members of configuration $c(k-1)$ have learned about configuration $c(k)$.

Definition 10.7 *Define the* gc-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 $i \in members(c(k-1))$, $cmap(\ell)_i \neq \bot$.*

The first lemma shows that soon after a configuration is installed, every node that joined sufficiently long ago learns about the new configuration. More specifically, we show that a configuration map containing the new configuration $c(k)$ becomes mainstream soon after the new configuration is installed.

Lemma 10.8 *Assume that configuration $c(k)$ is installed at time $t \geq 0$. Then there exists a configuration map 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 10.3 that $\mathit{cmap}(k)_j$ is mainstream after time $\max(t, T_{GST}) + 2d$. □

Since our goal is to show that configuration $c(k-1)$ is viable for sufficiently long after the gc-ready(k) event, and since configuration-viability only guarantees viability for a period of time after configuration $c(k)$ is installed, we need to show that the gc-ready(k) event occurs soon after configuration $c(k)$ (and all prior configurations) are installed. The next lemma shows exactly this, i.e., that a gc-ready(k) event occurs soon after all configurations with index smaller than k have been installed. The key is to show that every member of configuration $c(k-1)$ has learned about configuration $c(k)$, as well as every prior configuration. We demonstrate that, for each configuration $c(\ell)$ where $\ell \leq k$, a configuration map containing the configuration $c(\ell)$ is mainstream by no later than $\max(t, T_{GST}) + 2d$, and thus by time $\max(t, T_{GST}) + 6d$, the gc-ready(k) event occurs. Notice that in this case, the $6d$ delay arises from the fact that in Theorem 10.6, we can only determine that a configuration map is mainstream after time $T_{GST} + 6d$.

Lemma 10.9 *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* gc-ready(k) *occurs by time* $\max(t, T_{GST}) + 6d$.

Proof. Recall that gc-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 10.8 that there is some cm where $cm(\ell) \neq \bot$ that is mainstream after $\max(t_\ell, T_{GST}) + 2d$. We conclude from Theorem 10.6 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 10.2 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. □

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

Corollary 10.10 *For some $k \geq 0$, assume that no* gc-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 10.9, an gc-ready$(k+1)$ event occurs in α, contradicting the hypothesis. □

Finally, we show that if a gc-ready$(k+1)$ event does occur, then configuration $c(k)$ remains viable until at least $16d$ after the gc-ready$(k+1)$ event. This relies on the fact that the gc-ready$(k+1)$ event occurs within $6d$ of the time at which all configurations $\leq c(k+1)$ have been installed, and the fact that by assumption configuration $c(k)$ is viable for at least time $22d$ after configuration $c(k+1)$ is installed.

Theorem 10.11 *For some $k \geq 0$, assume that* gc-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 10.9 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. □

10.4 Garbage collection

In this subsection, we analyze the performance of garbage-collection operations. The main result of this section, Theorem 10.17, shows that every garbage-collection operation completes within $4d$ time. The key challenge lies in showing that every configuration included in a given garbage collection remains viable; if the viability of some configuration expires, then the garbage collection will never terminate. Thus, in the process, we show that each configuration is garbage collected soon after the next configuration is installed.

An important corollary is that every configuration that remains when a read/write operation phase begins remains viable for at least time $3d$, i.e., for sufficiently long for the phase to complete. This is shown in Corollary 10.18, which relies on: (1) the fact that a garbage collection for configuration $c(k-1)$ begins soon after the gc-ready(k) event, (2) the conclusion that the garbage-collection operation completes within time $4d$, and (3) the configuration viability hypothesis.

The main 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 10.12 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 $\max(time(\rho), T_{GST}) + 4d$.

In order to ensure that configurations remain viable, we have to show that each configuration is garbage collected soon after a new configuration is installed. The proof proceeds through the following steps. First, we describe some circumstances under which a garbage collection operation begins, for example, that a new configuration is avilable and that no garbage collection is ongoing. Next we show that, if we assume the gc-completes hypothesis, then within $8d$ of a gc-ready$(k+1)$ event, configuration $c(k)$ has been garbage collected (along with all prior configurations). This argument essentially acts as the inductive step in the main proof. The third step is to show that if the gc-completes hypothesis holds, then configurations remain viable for sufficiently long. Finally, we show the main result, that is, that every garbage-collection operation completes within $4d$ time.

The first step of the proof is to 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 10.13 *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. $\forall \ell < k : s.cmap(\ell)_i \neq \perp$: By the definition of gc-ready(k), Part (ii), we know that for all $\ell \leq k$, for all non-failed $j \in members(c(k-1))$, $cmap(\ell)_j \neq \perp$ immediately after the gc-ready(k) event. Node i satisfies both requirements, and later updates do not change this fact.
5. $s.cmap(k-1)_i \in C$: We have already shown (Part 4) that $s.cmap(k-1)_i \neq \perp$. By Assumption 6, $s.cmap(k-1)_i \neq \pm$.
6. $s.cmap(k)_i \in C$: We have already shown (Part 4) that $s.cmap(k)_i \neq \perp$. Since $s.cmap_i \in Usable$ (Invariant 1), and since $s.cmap(k-1)_i \neq \pm$ (Assumption 6), we can conclude that $s.cmap(k)_i \neq \pm$.

Since enabled events occur in zero time (by assumption), we conclude that a 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 8.1), we conclude that $k' \geq k$. □

Next, we show that if we assume the gc-completes hypothesis, then within $8d$ after a gc-ready$(k+1)$ event, some node in configuration $c(k)$ has already removed configuration $c(k)$. Essentially, this lemma relies on Lemma 10.13 to show that some garbage collection is started, and the gc-completes hypothesis to show that the garbage collection completes.

Lemma 10.14 *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* gc-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 10.11 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 gc-ready$(k+1)$ event:

- *Case 1:* Assume that $gc.phase_i = idle$ immediately after the gc-ready$(k+1)$ event:
  Notice that all the conditions of Lemma 10.13 are satisfied: (1) a gc-ready$(k+1)$ event occurs at time t_1; (2) let ρ be the gc-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 later updates do not invalidate this fact. Thus we conclude that i performs a gc$(k')_i$ event for some $k' > k$ at time t_1.

- *Case 2:* Assume that $gc.phase_i = active$ immediately after the gc-ready$(k+1)$ event:
  This implies that some event $\rho =$ 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 gc-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 10.13: (1) a gc-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 later updates do not invalidate this fact. We conclude that i performs a gc$(k')_i$ event for some $k' > k$ at time t_{ack}.

In either case, i performs a 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 gc-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 10.14 to show that a configuration with index k is garbage collected soon after the gc-ready$(k+1)$ event, and also Theorem 10.11 to show that configuration $c(k)$ remain viable long enough after the gc-ready$(k+1)$ event; finally, it uses Theorem 10.6 to show that once a configuration is removed, every other node learns about it sufficiently rapidly.

Lemma 10.15 *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 10.10 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 10.11 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 will 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 10.14 demonstrates that for some $j \in members(c(k))$ that does not fail at or before time $\max(t_{ready}, T_{GST}) + 10d$, the following holds: $cmap(k)_j = \pm$ 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 10.3 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 10.6 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 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. There are two cases, depending on whether the garbage collection begins before or after the network stabilizes; in either case, since sufficient quorums remain viable, the operation completes in time $4d$.

Lemma 10.16 *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. The proof proceeds by induction, with the gc-completes hypothesis as the inductive hypothesis. Lemma 10.16 shows that we need only demonstrate that appropriate quorums remain viable, and Lemma 10.15 guarantees the requisite viability.

Theorem 10.17 *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 gc-ack$_j$ by time $\max(t', T_{GST}) + 4d$.

We examine the inductive step: Consider the $(n+1)^{\text{st}}$ 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 10.16 shows that proving the following is sufficient: for every configuration $\ell \in removal\text{-}set(\gamma) \cup \{k\}$ 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 $\ell \in \textit{removal-set}(\gamma) \cup \{k\}$.

  We now apply Lemma 10.15: Notice that $\textit{cmap}(\ell) \in C$ at time t_{gc}, by the choice of ℓ in the $\textit{removal-set}(\gamma)$. 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 before time t_{gc} is necessarily one of the first n garbage collections in α. Thus we conclude from Lemma 10.15 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 conclude this subsection with a corollary that shows the following: as long as a configuration is in use by an $\textit{cmap}$, it remains viable. This is simply an unconditional version of Lemma 10.15, that is, a version that does not depend on assuming the gc-completes hypothesis. This corollary is critical in showing that read and write operations complete efficiently, as it ensures that quorums remain viable throughout the operation.

Corollary 10.18 *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 $k \geq 0$, $\textit{cmap}(k)_i \in C$ 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 10.15 that is not assumed here is that α satisfies the gc-completes hypothesis for t. This follows immediately from Theorem 10.17, implying the desired conclusion. Alternatively, if $t \leq \ell\textit{time}(\alpha') + 13d$, the claim follows immediately from configuration-viability. □

10.5 Reads and writes

In this section, we prove the main result which states that every read or write operation completes in time $8d$. 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 10.19 *Assume that a $\mathsf{report}(c(\ell))_i$ event occurs at time t for some $i \in \textit{members}(c(\ell))$ and some index ℓ. Then there exists a configuration map 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 $W \in \textit{write-quorums}(c(\ell))$ such that for every node $j \in W$, a $\mathsf{report}(c(\ell))_j$ occurs in α prior to the first $\mathsf{recon}(c(\ell), *)$ event. By configuration-viability, there exists some read-quorum $R \in \textit{read-quorums}(c(\ell))$ 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 i of the configuration $c(\ell)$ at time t, and since both i and j are members of $c(\ell)$, we can conclude that by time $\max(t, T_{GST}) + 2d$, $\textit{cmap}(\ell)_j \neq \bot$. Let $cm = \textit{cmap}_j$ 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 10.3 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 9.7: we show that each phase of a read or write operation is interrupted at most once by a new configuration due to recon-spacing; and Corollary 10.18 ensures that configurations remain viable for sufficiently long.

Theorem 10.20 *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*[6]*. Then node i completes the read or write operation by time $t + 8d$.*

Proof. Let $c_0, c_1, c_2, \ldots$ denote the (possibly) infinite sequence of successive configurations decided upon in α. For each $k \geq 0$, let π_k be the first $\mathsf{recon}(c_k, c_{k+1})_*$ event in α, if such an event occurs; let i_k be the node at which this occurs; let ϕ_k be the corresponding, preceding $\mathsf{report}(c_k)_{i_k}$ event.

We show that the time for each phase of the read or write operation is no more than $4d$ – 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 $\textit{time}(\psi) > \textit{time}(\phi_0) + 8d$, that is, that ψ occurs more than $8d$ time after the $\mathsf{report}(0)_{i_0}$ event: We have that $\textit{time}(\psi) \geq t$, and $t > \textit{time}(\mathsf{join\text{-}ack}_i) + 8d$, by assumption. Also, $\textit{time}(\mathsf{join\text{-}ack}_i) \geq \textit{time}(\mathsf{join\text{-}ack}_{i_0})$. Furthermore, $\textit{time}(\mathsf{join\text{-}ack}_{i_0}) \geq \textit{time}(\phi_0)$, 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 $\textit{time}(\psi) > \textit{time}(\phi_0) + 8d$.

[6] 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.

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 9.7.) Fix any $\ell \leq k$. We apply Lemma 10.19 to conclude that there exists a configuration map 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 10.6 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 10.18 shows that the configurations remain viable for sufficiently long. □

11 Conclusions

In this paper, we have presented RAMBO, an algorithm for implementing a reconfigurable read/write shared memory in an asynchronous message-passing system. RAMBO guarantees that read and write operations will always be executed atomically, regardless of network instability or timing asynchrony, and it guarantees that when the network is stable, operations will complete rapidly, subject to some reasonable assumptions.

RAMBO's Goals

An important goal of RAMBO was to decouple the implementation of read/write operations from the problem of configuration management. (As discussed in Section 2.2, this is in direct contrast to the more integrated "replicated state machine approach," in which both read/write operations and reconfigurations are implemented via the same agreement mechanism.) We believe that decoupling these two problems can result in better performance, especially in networks that experience highly variable message delays, such as wide-area networks and wireless networks. In such systems, reconfiguration may be delayed due to unpredictable asynchronies; in RAMBO, read and write operations can continue, unaffected by network instability. We also believe that decoupling read/write operations from reconfiguration may allow for the independent optimization of these two services; it may be possible that better performance is achieved by optimizing these services separately, rather than together.

Another important goal of RAMBO was to allow the client complete flexibility in choosing new configurations. This flexibility should allow clients to more easily choose good configurations, for example, those that will have good viability for a long period of time. Or, as another example, clients may choose configurations based on geography; in [10], for example, each configuration consists of a set of nodes that lie in a specific region.

Finally, we have described RAMBO in an abstract and non-deterministic manner, which allows RAMBO to be easily adapted for use in a given system. When implementing RAMBO, there is significant flexibility in when messages are sent, to whom messages are sent, how nodes join the system, and how reconfiguration is implemented. This flexibility, along with the unrestricted choice of configurations, has led to several extensions and implementations of RAMBO, as discussed in Section 2.3 (for example, [10,17,29,47–49,57]). In this way, we see RAMBO as an architectural template for future systems.

Technical Challenges

Perhaps the key technical challenge addressed by RAMBO is the problem of coordinating concurrent read/write operations with the ongoing configuration management process. For example, even as a read or write operation attempts to access the system, the configuration management process may be reconfiguring the system, removing old participants and introducing new participants. Unlike in other approaches, read/write operations are not halted when the system is reconfigured. Algorithmically, this has two main implications. First, read and write operations much be flexible in choosing which configurations to use, adopting any and every new configuration that is discovered during an operation. Second, garbage-collection operations must ensure that ongoing operations discover any new configuration that is installed, prior to removing an old configuration; at the same time, they have to propagate information from one configuration to the next. Together, these two techniques ensure that changes in the underlying set of participants has no effect on read and write operations.

A second technical challenge addressed by RAMBO is the problem of tolerating network instability. RAMBO ensures good performance of read and write operations during periods of network stability, and guarantees rapid stabilization after periods of network instability. This is achieved primarily through the rapid dissemination of information, and the rapid adoption of any new information. Gossip messages are sent continuously and frequently, and this ensures that nodes receive up-to-date information relatively quickly. (This may be contrasted to an algorithm in which messages are sent only during read and write operations, for example.) Moreover, as soon as a node learns about a new configuration, or the removal of an old configuration, it updates it view of the system to reflect this new information. Thus, changes to the system are rapidly integrated into local views, allowing for nodes to catch up rapidly when the system stabilizes.

In fact, an important contribution of this paper is the performance analysis of RAMBO, along with the techniques and tools developed to facilitate this analysis. As part of this analysis, we developed a set of hypotheses that define the dynamics of the system and bound the instability of the network during good intervals. For example, we introduced the idea of *configuration viability* to capture the notion that some quorums from each configuration need to survive until the next configuration has been installed. We believe that these hypotheses may be useful in analyzing other dynamic systems.

Open Questions and Ongoing Research

The results in this paper raise several interesting research questions. The first natural question is whether the RAMBO approach can be applied to other types of data objects. While RAMBO implements a read/write distributed shared memory, it may be possible to implement stronger distributed objects, such as sets, queues, snapshot objects, etc. In fact, we believe that it is possible to adopt any quorum-based algorithm to fit the RAMBO paradigm. (For a more powerful object, such as a compare-and-swap, there may be fewer advantages from decoupling the configuration management process; however, the modularity may still be beneficial.)

A second important question relates to the strategy for choosing new configurations. Any quorum-based algorithm can be only as robust as the underlying configuration viability. Thus, it is imperative that clients choose good configurations that have sufficient viability, and that clients initiate reconfiguration sufficiently frequently to ensure that configurations always remain available. We believe that as long as periods of instability are sufficiently low, and as long as the rate of failures is sufficiently low, then it is possible to ensure good configuration viability.

The choice of configurations, along with many other aspects of RAMBO implementations, depends significantly on the underlying environment in which the system is executing. For example, in a peer-to-peer distributed hash table (see [49]), the configurations may depend on how data is distributed in the network; in a wireless ad hoc network, the configurations may depend on the geographic distribution of devices (see [10]). And, in many other ways as well, the RAMBO algorithm contains only the outlines for a real implementation: How often should nodes exchange information? With whom should a node exchange information? How much information should they send in each message? How should nodes join the system? Each of these questions depends greatly on the underlying application. Answering these questions will require implementing RAMBO in various contexts, optimizing for the most important concerns, and understanding the trade-offs that arise in translating the RAMBO architectural template into a real system.

Overall, we anticipate that the approach presented in this paper will continue to influence follow on research and practical implementations of robust algorithms for highly dynamic systems.

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

References

[1] A. El Abbadi, D. Skeen, and F. Cristian. An efficient fault-tolerant protocol for replicated data management. In *Proceedings of the Symposium on Principles of Databases*, pages 215–228, 1985.

[2] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *Transactions on Database Systems*, 14(2):264–290, 1989.

[3] I. Abraham and D. Malkhi. Probabilistic quorums for dynamic systems. *Distributed Computing*, 18(2):113–124, 2005.

[4] D. Agrawal and A. El Abbadi. Resilient logical structures for efficient management of replicated data. In *Proceedings of the International Conference on Very Large Data Bases*, pages 151–162, 1992.

[5] J. R. Albrecht and Y. Saito. RAMBO for dummies. Technical Report HPL-2005-39, Hewlett-Packard, 2005.

[6] L. Alvisi, D. Malkhi, E. T. Pierce, and M. K. Reiter. Fault detection for byzantine quorum systems. *Transactions on Parallel and Distributed Systems*, 12(9):996–1007, 2001.

[7] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. Moser. Robust and efficient replication using group communication. Technical Report 1994-20, Hebrew University, 1994.

[8] Y. Amir and A. Wool. Evaluating quorum systems over the internet. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 26–35, 1996.

[9] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[10] J. Beal and S. Gilbert. RamboNodes for the metropolitan ad hoc network. In *Proceedings of the Workshop on Dependability Issues in Wireless Ad Hoc Networks and Sensor Networks*, 2004.

[11] M. Bearden and R. P. Bianchini Jr. A fault-tolerant algorithm for decentralized on-line quorum adaptation. In *Proceedings of the International Symposium on Fault-Tolerant Computing Systems*, pages 262–271, Munich, Germany, 1998.

[12] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[13] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[14] B. Charron-Bost and A. Schiper. Improving fast paxos: being optimistic with no overhead. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, pages 287–295, 2006.

[15] G. Chockler, S. Gilbert, V. Gramoli, P. M. Musial, and A. A. Shvartsman. Reconfigurable distributed storage for dynamic networks. In *Proceedings of the International Conference on Principles of Distributed Systems*, pages 214–219, 2005.

[16] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, 1985.

[17] S. Dolev, S. Gilbert, N. A. Lynch, A. A. Shvartsman, and J. L. Welch. Geoquorums: implementing atomic memory in mobile ad hoc networks. *Distributed Computing*, 18(2):125–155, 2005.

[18] B. Englert and A. A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of the International Conference on Distributed Computer Systems*, pages 454–463, 2000.

[19] A. Fekete, N. A. Lynch, and A. A. Shvartsman. Specifying and using a partitionable group communication service. *Transaction on Computer Systems*, 19(2):171–216, 2001.

[20] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[21] H. Garcia-Molina and D Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, 1985.

[22] C. Georgiou, P. M. Musial, and A. A. Shvartsman. Developing a consistent domain-oriented distributed object service. In *Proceedings of the International Symposium on Network Computing and Applications*, pages 149–158, 2005.

[23] C. Georgiou, P. M. Musial, and A. A. Shvartsman. Long-lived RAMBO: Trading knowledge for communication. *Theoretical Computer Science*, 383(1):59–85, 2007.

[24] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Symposium on Operating Systems Principles*, pages 150–162, 1979.

[25] S. Gilbert. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. Master's thesis, MIT, 2003.

[26] S. Gilbert, N. A. Lynch, and A. A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 259–268, 2003.

[27] K. Goldman and N. A. Lynch. Quorum consensus in nested transaction systems. *Transactions on Database Systems*, 19(4):537–585, 1994.

[28] V. Gramoli. RAMBO III: Speeding-up the reconfiguration of an atomic memory service in dynamic distributed system. Master's thesis, Université Paris Sud–Orsay, 2004.

[29] V. C. Gramoli, P. M. Musial, and A. A. Shvartsman. Operation liveness and gossip management in a dynamic distributed atomic data service. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pages 206–211, 2005.

[30] M. Herlihy. *Replication Methods for Abstract Data Types*. PhD thesis, Massachusettes Institute of Technology, 1984.

[31] M. Herlihy. Dynamic quorum adjustment for partitioned data. *Transactions on Database Systems*, 12(2):170–194, 1987.

[32] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *Transactions on Database Systems*, 15(2):230–280, 1990.

[33] D. K. Kaynar, N. A. Lynch, R. Segala, and F. Vaandrager. The theory of timed I/O automata. Technical Report MIT-LCS-TR-917a, MIT, 2004.

[34] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Hebrew University, Jerusalem, 1994.

[35] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *Proceedings of the Symposium on Principles of Distributed Domputing*, pages 68–76, 1996.

[36] K. M. Konwar, P. M. Musial, N. C. Nicolaou, and A. A. Shvartsman. Implementing atomic data through indirect learning in dynamic networks. In *Proceedings of the International Symposium on Network Computing and Applications*, pages 223–230, 2007.

[37] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[38] L. Lamport. The part-time parliament. *Transactions on Computer Systems*, 16(2):133–169, 1998.

[39] L. Lamport. Fast paxos. Technical Report MSR-TR-2005-12, Microsoft, 2005.

[40] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[41] M. Liu, D. Agrawal, and A. El Abaddi. On the implementation of the quorum consensus protocol. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pages 318–325, 1995.

[42] E. Y. Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 63–71, 1997.

[43] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[44] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 272–281, 1997.

[45] N. A. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the International Symposium on Distributed Computing*, pages 173–190, 2002.

[46] D. Malkhi and M. K. Reiter. Byzantine quorum systems. In *Proceedings of the Symposium on Theory of Computing*, pages 569–578, 1997.

[47] P. M. Musial. *From High Level Specification to Executable Code: Specification, Refinement, and Implementation of a Survivable and Consistent Data Service for Dynamic Networks*. PhD thesis, University of Connecticut, Storrs, 2007.

[48] P. M. Musial and A. A. Shvartsman. Implementing a reconfigurable atomic memory service for dynamic networks. In *Proceedings of the International Parallel and Distributed Processing Symposium*, page 208b, 2004.

[49] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: a fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-LCS-TR-993, MIT, 2005.

[50] M. Naor and U. Wieder. Scalable and dynamic quorum systems. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 114–122, 2003.

[51] M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *Journal on Computing*, 27(2):423–447, 1998.

[52] D. Peleg and A. Wool. The availability of quorum systems. *Information and Computation*, 123(2):210–223, 1995.

[53] D. Peleg and A. Wool. How to be an efficient snoop, or the probe complexity of quorum systems. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 290–299, 1996.

[54] R. De Prisco, A. Fekete, N. A. Lynch, and A. A. Shvartsman. A dynamic primary configuration group communication service. In *Proceedings of the International Symposium on Distributed Computing*, pages 64–78, 1999.

[55] R. De Priso, B. Lampson, and N. Lynch. Revisiting the PAXOS algorithm. *Theoretical Computer Science*, 243(1–2):35–91, 2000.

[56] S. Rangarajan and S. Tripathi. A robust distributed mutual exclusion algorithm. In *Proceedings of the International Workshop on Distributed Algorithms*, pages 295–308, 1991.

[57] Y. Saito, S. Frølund, A. C. Veitch, A. Merchant, and S. Spence. Fab: building distributed enterprise disk arrays from commodity components. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–58, 2004.

[58] B. A. Sanders. The information structure of distributed mutual exclusion algorithms. *Transactions on Computer Systems*, 5(3):284–299, 1987.

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