Rambo II: Rapidly Reconfigurable Atomic Memory for Dynamic Networks *

Seth Gilbert† Nancy Lynch‡ Alex Shvartsman§

January 9, 2004

Abstract

Future civilian rescue and military operations will depend on a complex system of communicating devices that can operate in highly dynamic environments. In order to present a consistent view of a complex world, these devices will need to maintain data objects with atomic (linearizable) read/write semantics.

Lynch and Shvartsman have recently developed a reconfigurable atomic read/write memory algorithm for such environments [12, 13] This algorithm, called Rambo, guarantees atomicity for arbitrary patterns of asynchrony, message loss, and node crashes. Rambo installs new configurations lazily, transferring data from old configurations to new configurations using a background information transfer task. That task handles configurations sequentially, transferring information from each configuration to the next.

This paper presents a new algorithm, Rambo II, that implements a radically different approach to installing new configurations: instead of operating sequentially, the new algorithm reconfigures "aggressively", transferring information from old configurations in parallel. This improvement substantially reduces the time necessary to remove obsolete configurations, which in turn substantially increases the fault-tolerance. This paper presents a formal specification of the new algorithm, a correctness proof, and a conditional analysis of its performance. Preliminary empirical studies performed using LAN implementations of Rambo and the new algorithm illustrate the advantages of the new algorithm.

---

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

†Massachusetts Institute of Technology, Laboratory for Computer Science, 200 Technology Square, NE43-371, Cambridge, MA 02139, USA. Email: `sethg@theory.lcs.mit.edu`.

‡Massachusetts Institute of Technology, Laboratory for Computer Science, 200 Technology Square, NE43-365, Cambridge, MA 02139, USA. Email: `lynch@theory.lcs.mit.edu`. The work of this author was additionally supported by NSF Grant 9804665.

§Department of Computer Science and Engineering, 191 Auditorium Road, Unit 3155, University of Connecticut, Storrs, CT 06269 and Massachusetts Institute of Technology, Laboratory for Computer Science, 200 Technology Square, NE43-363, Cambridge, MA 02139, USA. Email: `alex@theory.lcs.mit.edu`. The work of this author was additionally supported by the NSF Career Award 9984778 and NSF Grant 9988304.

Contents

1 Introduction **3**

2 The Original Rambo Algorithm **6**

3 Formal Specification of RAMBO II **10**

4 Notation and Basic Lemmas **14**
- 4.1 Good Executions . . . 14
- 4.2 Notational conventions . . . 15
- 4.3 Configuration map invariants . . . 17
- 4.4 Phase guarantees . . . 20

5 Atomic Consistency **22**
- 5.1 Behavior of configuration upgrade . . . 23
- 5.2 Behavior of a read or a write following a configuration upgrade . . . 26
- 5.3 Behavior of sequential reads and writes . . . 27
- 5.4 Atomicity . . . 30

6 Reconfiguration Service **31**
- 6.1 Reconfiguration Service Specification . . . 31
- 6.2 Reconfiguration Service Implementation . . . 32
- 6.3 Consensus services . . . 33
- 6.4 Recon automata . . . 34

7 Conditional Performance Analysis **36**
- 7.1 Definitions . . . 36
- 7.2 Limiting Nondeterminism . . . 37
- 7.3 The Behavior of the Environment . . . 37
  - 7.3.1 Normal Timing Behavior from Some Point Onward . . . 38
  - 7.3.2 Configuration–Viability . . . 38
  - 7.3.3 Recon-Spacing . . . 39
  - 7.3.4 Join-Connectivity . . . 39
  - 7.3.5 Recon-Readiness . . . 40
  - 7.3.6 Upgrade-Readiness . . . 40
  - 7.3.7 Fixed Parameters . . . 40
- 7.4 Basic Lemmas . . . 41
- 7.5 Propagation of Information . . . 42
- 7.6 Upgrade-Ready Viability . . . 47
- 7.7 Configuration-Upgrade Latency Results . . . 50
- 7.8 Read-Write Latency Results . . . 61

8 Implementation and Preliminary Evaluation **64**

9 Conclusion and Open Problems **66**

1 Introduction

Future large scale civilian rescue and military deployment operations will involve large numbers of communication and computing devices operating in highly dynamic network substrates. Successful coordination and marshaling of human resources and equipment involves collecting information about a complex real-world situation using sensors and input devices, gathering the information in survivable repositories, and providing appropriate and coherent information to the stakeholders.

Data objects with atomic (linearizable) read/write semantics commonly occur in such settings. Replication of objects is a prerequisite for fault-tolerance and availability, and with replication comes the need to maintain consistency. Additionally, in dynamic settings where participants may join and leave the environment, may fail, and where the physical objects migrate, one needs to be able to effectively move the corresponding data objects from one set of data owners to another.

Lynch and Shvartsman developed a reconfigurable atomic read/write memory algorithm for dynamic networks [12, 13]. The algorithm, called RAMBO, guarantees atomicity for arbitrary patterns of asynchrony, message loss, and node crashes. Conditional performance analysis of the algorithm shows that when the environment timing stabilizes, when failures are within specific parameters, and when the reconfigurations are not frequent and not bursty, then read and write operations have small latency bounded in terms of the maximum message delay and the periodic gossip interval. However when the reconfigurations are frequent or bursty, this algorithm may perform poorly because of the inherently sequential processing of the new configurations once they become determined by the algorithm. In particular, the number of configurations maintained by the algorithm may grow without bound, leading to the unbounded number of messages necessary in processing the read and write operations. Such situations may arise due to failures or asynchrony, yet these are not the only reasons. Even in synchronous failure-free environments the world dynamics may require that frequent reconfigurations are performed to keep track of the rapidly moving physical objects or rapidly changing set of stakeholders.

This paper presents a new algorithm, RAMBO II, integrated with RAMBO, that implements a radically different approach to installing new configurations: instead of operating sequentially, the new algorithm reconfigures "aggressively", transferring information from old configurations in parallel. This improvement substantially reduces the time necessary to process new configurations and to remove obsolete configurations from the system, which in turn substantially increases fault-tolerance. This is due to the fact that once a configuration is removed, the system no longer depends on it, and as soon as the configuration is removed, it is allowed to fail. The process executing the new algorithm achieves a linear speed-up in the number of old configurations known to the process. For example, our conditional performance analysis shows that if a process knows about a sequence of h configurations, then the it can eliminates all but one of these configurations in time $O(1)$, as compared to the original RAMBO, where this takes $\Theta(h)$ time. Additionally, the new algorithm reduces the number of messages necessary to process these configurations

This paper presents a formal specification of the new algorithm, a correctness proof, and a conditional analysis of its performance. Preliminary empirical studies performed using LAN implementations of RAMBO and the new algorithm illustrate the advantages of the new algorithm.

Background. Starting with the work of Gifford [6] and Thomas [18], intersecting collections of sets found use in several algorithms providing consistent data in distributed settings. Depending on the algorithm and its setting, such collections of sets, called quorums when any two have non-empty intersection, represent either sets of processors or their knowledge. Upfal and Wigderson [19] use majority sets of readers and writers to emulate shared memory in a distributed setting.

Vitányi and Awerbuch [20] implement multi-writer/multi-reader registers using matrices of single-writer/single-reader registers where the rows and the columns are written and respectively read by specific processors. Attiya, Bar-Noy and Dolev [1] use majorities of processors to implement single-writer/multi-reader objects in message passing systems. Such algorithms assume a static processor universe and rely on static static quorum systems.

In long-lived systems where processors may dynamically join and leave the system, it is important to reconfigure a quorum system to adapt it to the new set of processors [8, 4, 7, 17]. Prior approaches required that the new quorum system include processors from the old quorum system. This is stated as a static constraint on the quorum system that needs to be satisfied during or even before the reconfiguration. In our work on reconfigurable atomic memory [15, 5, 12] we replace the *space-domain* requirement on successive quorum system intersections with the *time-domain* requirement that some quorums from the old and the new system are involved in the reconfiguration algorithm. Such systems are more dynamic because they allow for more choices of new quorum systems and do not require that successive configurations intersect.

Reconfiguration in Highly Dynamic Settings. Lynch and Shvartsman's earlier algorithms [15, 5] allowed a single distinguished process to act as the quorum system reconfigurer. The advantage of the single-reconfigurer approach is its relative simplicity and efficiency: any process maintains at most two configurations, the current configuration and the proposed new configuration. The disadvantage of the single reconfigurer is that it is a single point of failure – no further reconfiguration is possible if the reconfigurer fails.

The RAMBO algorithm [12, 13] removed the requirement of having a single reconfigurer, thus enabling any process within its own current configuration to begin reconfiguration to a new quorum system supplied by the environment. The algorithm implements atomic shared memory suitable for use in highly dynamic settings, and it guarantees atomicity in any asynchronous execution and in the presence of arbitrary process and network failures. However the multiple-reconfigurer approach introduces the problem of maintaining multiple configurations and removing old configurations from the system. RAMBO implements a sequential "garbage-collection" algorithm where processes remove obsolete configurations one-at-a-time. Configuration removal requires that information is propagated from the earliest known configuration to its successor. Since arbitrarily many new configurations may be introduced this leads to an unbounded number of old configurations that need to be sequentially removed.

The environment may introduce new configurations for several reasons: (*i*) due to failures and network instability that endanger installed configurations, (*ii*) due to the mobility of the physical objects represented by the abstract memory objects and the mobility of the processes maintaining the object replicas, and (*iii*) due to the need to rebalance loads on processes within installed configurations. Frequent or bursty reconfiguration can substantially increase the number of installed configurations and, since a process performing a read or a write operation potentially needs to contact quorums in all configurations known to it, this leads to the corresponding increase in the number of messages needed to perform the operation.

The New Algorithm. The primary contribution of this paper is a new algorithm for reconfigurable atomic memory, based on the original RAMBO, that implements an aggressive configuration-replacement protocol where any locally-known contiguous sequence of configurations is replaced by the last configuration in the sequence. The removal of the old configurations is done in parallel, while preserving all other properties of the original RAMBO. Specifically, we maintain a loose coupling between the reconfiguration algorithms and the original RAMBO algorithms implementing the read and write operations.

In order to achieve availability in the presence of failures, the objects are replicated at several network locations. In order to maintain memory consistency in the presence of small and transient changes, the algorithm uses *configurations*, each of which consists of a set of *members* plus sets of *read-quorums* and *write-quorums*. In order to accommodate larger and more permanent changes, the algorithm supports *reconfiguration*, by which the set of members and the sets of quorums are modified. Such changes do not cause violations of atomicity. Any quorum configuration may be installed at any time—no intersection requirement is imposed on the sets of members or on the quorums of distinct configurations.

The algorithm is composed of a *main algorithm*, which handles reading, writing, and replacement of old configurations with a successor configuration, and a global configuration announcement service, *Recon*, which provides the main algorithm with a consistent sequence of configurations. Several configurations may be known to the algorithm at one time, and read and write operations can use them all without any harm.

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

The main algorithm provides a new configuration-replacement algorithm that removes old configurations while ensuring that their use is no longer necessary for maintaining consistency. Configuration-replacement also uses a two-phase strategy, where the first phase communicates in parallel with all old configurations being removed and the second phase communicates with a new configuration. A configuration-replacement operation ensures that both a read-quorum and a write-quorum of each old configuration learn about the new configuration, and that the latest value from all old configurations is conveyed to a write-quorum of the new configuration. The strength of the new algorithm is that it proceeds aggressively in parallel. An arbitrary number of old configurations can be replaced in constant time (assuming bounded message latency and non-failure of active configurations).

The configuration announcement service is implemented by a distributed algorithm that uses distributed consensus to agree on the successive configurations. Any member of the latest configuration c may propose a new configuration at any time; different proposals are reconciled by an execution of consensus among the members of c. Consensus is, in turn, implemented using a version of the Paxos algorithm [9], as described formally in [3]. Although such consensus executions may be slow—in fact, in some situations, they may not even terminate—they do not cause any delays for read and write operations.

All services and algorithms, and their interactions, are specified using I/O automata. We show correctness (atomicity) of the algorithm for arbitrary patterns of asynchrony and failures. On the other hand, we analyze performance *conditionally*, based on certain failure and timing assumptions. For example, assuming that gossip and configuration-replacement occur periodically, and that quorums of active configurations do not fail, we show that read and write operations complete within time $8d$, where d is the maximum message latency. Note that the original RAMBO algorithm also had to assume also that garbage-collection is able to keep up—this assumption is not necessary in the new algorithm due to the new configuration replacement algorithm. For the configuration replacement algorithm we show that any number of configurations can be replaced by their successor in constant time.

At the same time, all the performance results of the original RAMBO algorithm still hold; in instances where the network is reliable and timely throughout the execution, the bounds described in the previous RAMBO papers [12, 13] still hold.

Implementations of RAMBO and RAMBO II on a LAN are currently being completed [16]. Preliminary empirical studies performed using this implementation illustrate the advantages of the new algorithm over the previous one.

Document Structure. In Section 2 we describe the original RAMBO algorithm of Lynch and Shvartsman, and then in Section 3 present and discuss the formal specification of RAMBO II. In Section 4 we present some notation, and restate some basic lemmas, only slightly modified from RAMBO. In Section 5 we prove that the new algorithm guarantees atomic consistency. In Section 6 we present the reconfiguration service. In Section 7 we analyze the performance of RAMBO II, and discuss in detail the areas in which this algorithm improves over the original RAMBO algorithm. In Section 8 we discuss the preliminary performance results. Finally, in Section 9 we summarize the results, and areas for future research.

2 The Original Rambo Algorithm

In this section, we present the original RAMBO algorithm, on which the new algorithm RAMBO II is based. RAMBO is an algorithm designed to support read/write operations on an atomic shared memory.

In order to achieve fault tolerance and availability, RAMBO replicates data at several network locations. The algorithm uses *configurations* to maintain consistency in the presence of small and transient changes. Each configuration consists of a set of *members* plus sets of *read-quorums* and *write-quorums*. The quorum intersection property requires that every read-quorum intersect every write-quorum. Read and write operations are implemented as a two-phase protocol, in which each phase accesses a set of read or write quorums.

RAMBO supports *reconfiguration*, which modifies the set of members and the sets of quorums, thereby accommodating larger and more permanent changes without violating atomicity. In this way, failed nodes can be removed from active quorums, and newly joined nodes can be integrated into the system. Any quorum configuration may be installed at any time – no intersection requirement is imposed on the sets of members or on the quorums of distinct configurations.

The RAMBO algorithm consists of three kinds of automata:

- *Joiner* automata, which handle join requests,
- *Recon* automata, which handle reconfiguration requests, and generate a totally ordered sequence of configurations, and
- *Reader-Writer* automata, which handle read and write requests, manage garbage collection, and send and receive gossip messages.

In this paper, we discuss only the *Reader-Writer* automaton. The *Joiner* automaton is quite simple; it sends a join message when node i joins, and sends a join-ack message in response to join messages. The *Recon* automaton depends on a consensus service, implemented using Paxos [9], to agree on a total ordering of configurations. However, we assume that this total ordering exists, and therefore need not discuss this automaton any further. For more details of these two automata, see the original RAMBO paper [12, 13].

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

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

Figure 1: RAMBO(x): External signature

The external signature for RAMBO appears in Figure 1. The algorithm is specified for a single memory location, and extended to implement a complete shared memory. A client uses the join_i action to join the system. After receiving a $\mathsf{join\text{-}ack}_i$, the client can issue read_i and write_i requests, which results in $\mathsf{read\text{-}ack}_i$ and $\mathsf{write\text{-}ack}_i$ responses. The client can issue a recon_i request to propose a new configuration. Finally, the fail_i action is used to model node i failing.

The signature and state for the *Reader-Writer* automata is presented in Figure 2. The code for the *Reader-Writer* automata is presented in Figure 3. All three operations, *read*, *write*, and *garbage-collect*, are implemented using gossip messages. Unlike in many other algorithms, there are no directed messages specified in this algorithm; at no point does a given node, say i, decide to send a message specifically to node j. Instead, at regular intervals node i will non-deterministically send all of its public state to other nodes. Progress in an operation occurs when enough information has been exchanged. After initiating an operation, the automaton waits until it can be sure that it has shared state with enough other nodes (using gossip messages), and then declares the operation complete. The phase numbering regime, implemented using *pnum1* and *pnum2* is used to determine when enough communication has completed.

Every node maintains a *tag* and a *value* for the data object. Every new value is assigned a unique tag, with ties broken by process-ids. These tags are used to determine an ordering of the write operations, and therefore determine the value that a read operation should return.

Read and write operations require two phases, a query phase and a propagation phase, each of which accesses certain quorums of replicas. Assume the operation is initiated at node i. See Figure 5 for a summary of the two phases. First, in the query phase, node i contacts read quorums to determine the most recent available tag and value. Then, in the propagation phase, node i contacts write quorums. If the operation is a read operation, the second phase propagates the largest tag discovered in the query phase, and its associated value. If the operation is a write operation, node i chooses a new tag, strictly larger than every tag discovered in the query phase and propagates the new tag and the new value to the write quorums. Note that every operation accesses both read and write quorums.

During a phase of an operation, whenever node i receives a gossip message from node j, it compares the largest phase number j has received from i (by examining *pns*) to the local phase number when the operation began. If j initiated the gossip message after receiving a message from i sent after the phase began, then i adds j to the *acc* set. In effect, there has been a round-trip message sent from i to j back to i. Also, i then updates its *op.cmap* if necessary.

Garbage collection operations remove old configurations from the system. A garbage collection

Signature:

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

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

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

State:

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

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

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

Figure 2: $\mathit{Reader\text{-}Writer}_i$: Signature and state

operation involves two configurations: the old configuration being removed and the new configuration being established. See Figure 6 for a summary of the two phases. A garbage collection operation requires two phases, a query phase and a propagation phase. The first phase contacts a read-quorum and a write-quorum from the old configuration, and the second phase contacts a write-quorum from the new configuration.

Note that, unlike a read or write operation, the first phase of the garbage-collection operation must contact two types of quorums: a read-quorum and a write-quorum for the configuration being garbage-collected. This ensures that enough nodes are aware of the new configurations, and ensures that any ongoing read/write operations will include the new, larger, configuration.

The $cmap$ is a mapping from integer indices to configurations $\cup\{\bot, \pm\}$, that initially maps every index to $\bot$. The $cmap$ tracks which configurations are active, which are not defined, indicated by $\bot$, and which are removed, indicated by $\pm$. The total ordering on configurations determined by the *Recon* automata ensures that all nodes agree on which configuration is stored in each position in the array. We define $c(k)$ to be the configuration associated with index k.

The record op stores information about the current phase of an ongoing read or write operation, while gc stores information about an ongoing garbage collection operation. (A node can process

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

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

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

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

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

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

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

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

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

Figure 3: $\mathit{Reader\text{-}Writer}_i$: Read/write transitions

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

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

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

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

Figure 4: *Reader-Writer*$_i$: Rambo Garbage-collection transitions

read and write operations even when a garbage collection operation is ongoing.) The *op.cmap* subfield records the configuration map for an operation. This consists of the node's *cmap* when a phase begins, augmented by any new configurations discovered during the phase. A phase can complete only when the initiator has exchanged information with quorums from every non-removed configuration in *op.cmap*. The *pnum* subfield records the phase number when the phase begins, allowing the initiator to determine which responses correspond to the current phase. The *acc* subfield records which nodes from which quorums have responded during the current phase.

In Rambo, configurations go through three phases: proposal, installation, and upgrade. First, a configuration is *proposed* by a recon event. Next, if the proposal is successful, the *Recon* service achieves consensus on the new configuration, and notifies participants with decide events. When every non-failed member of the previous configuration has been notified, the configuration is *installed*. The configuration is *upgraded* when every configuration with a smaller index has been removed at some process in the system. Once a configuration has been upgraded, it is responsible for maintaining the data.

3 Formal Specification of Rambo II

In this section we present the new algorithm in detail, and discuss how it differs from the Rambo algorithm. The complete implementation, $\mathcal{S}$, is the composition of all the automata described—the

Operation initiated by read_i **or** $\mathsf{write}(v)_i$

Phase 1 :

- Node i communicates with a read-quorum from each configuration in $op.cmap$ in order to determine the largest value/tag pair.

Phase 2 :

- Node i communicates with a write-quorum from each configuration in in $op.cmap$ to notify it of the current largest value/tag pair (or the new value/tag pair, if it is a write operation).

Figure 5: Summary of two phase read or write operation

$Joiner_i$ and $Recon_i$ automata described in RAMBO, the new *Reader-Writer*$_i$ automaton described here, for all i, and all the channels – with all the actions that are not external actions of the RAMBO II specification hidden.

The key problem that prevents rapid stabilization in the original algorithm is the sequential nature of the configuration upgrade mechanism: in RAMBO, configurations are upgraded one at a time, in order. (Recall that in RAMBO, a configuration is upgraded when every configuration with a smaller index has been garbage collected.) Configuration $c(k)$ can be upgraded only if configuration $c(k-1)$ has previously been upgraded. This requirement arises from the need to ensure that information is preserved as configurations are changed. As in RAMBO, a configuration in RAMBO II is *upgraded* when every configuration with a smaller index has been removed at some process in the system. RAMBO II, however, implements a new reconfiguration protocol that can upgrade any configuration, even if configurations with smaller indices have not been upgraded. Unlike in RAMBO, then, there may be configurations that are not upgraded until they themselves are removed, at the same instant that some configuration with a larger index is upgraded.

After RAMBO II completes an upgrade operation for some configuration, all configurations with smaller indices can be removed. Thus a single upgrade operation in RAMBO II potentially has the effect of many garbage collection operations in RAMBO, each of which can only remove a single configuration. The name has been changed to emphasize the operation's active role in configuration management: configuration upgrade is an inherent part of preparing a configuration to assume responsibility for the data. The code for the new configuration management mechanism

Operation initiated by $\mathsf{gc}(k)_i$

Phase 1 :

- Node i communicates with a read-quorum from configuration $c(k)$ in order to determine the largest value/tag pair.
- Node i communicates with a write-quorum from configuration $c(k)$ in order to notify it of configuration $k+1$.

Phase 2 :

- Node i communicates with a write-quorum from configuration $c(k+1)$ to notify it of the current largest value/tag pair.

Figure 6: Summary of two phase garbage-collection operation

Signature:

As in RAMBO, with the following modifications:

Internal:

$\mathsf{cfg\text{-}upgrade}(k)_i,\ k \in \mathbb{N}^{>0}$
$\mathsf{cfg\text{-}upg\text{-}query\text{-}fix}(k)_i,\ k \in \mathbb{N}^{>0}$
$\mathsf{cfg\text{-}upg\text{-}prop\text{-}fix}(k)_i,\ k \in \mathbb{N}^{>0}$
$\mathsf{cfg\text{-}upgrade\text{-}ack}(k)_i,\ k \in \mathbb{N}^{>0}$

Configuration Management State:

As in RAMBO, with the following replacing the *gc* record:

upg, a record with fields:

$phase \in \{\mathsf{idle}, \mathsf{query}, \mathsf{prop}\}$, initially idle
$pnum \in \mathbb{N}$
$cmap \in CMap$,
acc, a finite subset of I
$target \in N$

Configuration Management Transitions:

Internal $\mathsf{cfg\text{-}upgrade}(k)_i$
Precondition:
$\neg failed$
$status = \mathsf{active}$
$upg.phase = \mathsf{idle}$
(A) $cmap(k) \in C$
$cmap(k-1) \in C^1$
(B) $\forall \ell \in \mathbb{N}, \ell < k : cmap(\ell) \neq \perp$
Effect:
$pnum1 \leftarrow pnum1 + 1$
(C) $upg \leftarrow \langle \mathsf{query}, pnum1, cmap, \emptyset, k \rangle$

Internal $\mathsf{cfg\text{-}upg\text{-}query\text{-}fix}(k)_i$
Precondition:
$\neg failed$
$status = \mathsf{active}$
$upg.phase = \mathsf{query}$
$upg.target = k$
(D) $\forall \ell \in \mathbb{N}, \ell < k : upg.cmap(\ell) \in C$
$\Rightarrow \exists R \in read\text{-}quorums(upg.cmap(\ell)) :$
$\exists W \in write\text{-}quorums(upg.cmap(\ell)) :$
(E) $R \cup W \subseteq upg.acc$
Effect:
$pnum1 \leftarrow pnum1 + 1$
(F) $upg.pnum \leftarrow pnum1$
$upg.phase \leftarrow \mathsf{prop}$
(G) $upg.acc \leftarrow \emptyset$

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

Internal $\mathsf{cfg\text{-}upgrade\text{-}ack}(k)_i$
Precondition:
$\neg failed$
$status = \mathsf{active}$
$upg.target = k$
$\forall \ell \in \mathbb{N}, \ell < k : \ cmap(\ell) = \pm$
Effect:
$upg.phase = \mathsf{idle}$

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

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

Figure 7: *Reader-Writer*$_i$: Configuration Management transitions

appears in Figure 7. All labeled lines in this section refer to the code therein.

We now describe in more detail the configuration upgrade operation, which is at the heart of RAMBO II. A configuration upgrade is a two-phase operation, much like the garbage-collection operation in RAMBO. See Figure 8 for a summary of the two phases. An upgrade operation is initiated at node i with a cfg-upgrade(k) event. When this happens, $cmap(k)$ must be defined, that is, must be a valid configuration $\in C$ (line A). Additionally, for every configuration $\ell < k$, $cmap(\ell)$ must be either $\in C$ or removed, that is, $\pm$ (line B).

We refer to configuration $c(k)$ as the *target* of the upgrade operation, and we refer to the set of configurations to be removed, $\{c(\ell) : \ell < k \ \wedge \ upg.cmap(\ell) \in C\}$, as the *removal-set* of the configuration upgrade operation. The configuration management mechanism guarantees that the *removal-set* consists of configurations with a contiguous set of indices.

As a result of the cfg-upgrade event, node i initializes its *upg* state (line C), and begins the query phase of the upgrade operation. In particular, node i stores its current *cmap* in *upg.cmap*, which records the configurations that are currently active. Only these configurations (and, in fact, only those with index smaller than k) matter during the operation; new configurations are ignored.

The query phase continues until node i receives responses from enough nodes. In particular, for every configuration $c(\ell)$ with index less than k in *upg.cmap*, there must exist a read-quorum, R, of configuration $c(\ell)$, and a write-quorum, W, of configuration $c(\ell)$ such that i has received a response (that is, a recent gossip message) from every node in $R \cup W$ (lines D–E).

When the query phase completes, a cfg-upg-query-fix event occurs. When this event occurs, node i then has the most recent tag and value discovered by operations using configurations with index smaller than k. Further, all configurations with indices smaller than k have been notified of configuration $c(k)$. Node i then reinitializes *upg* to begin the propagation phase (lines F–G).

The propagation phase continues until node i receives responses from a write-quorum in configuration $c(k)$. In particular, there must exist a write-quorum, W, of configuration $c(k)$, such that i has received a response from every node in W (line H).

When the propagation phase completes, a cfg-upg-prop-fix event occurs, which verifies the termination condition. At this point node i has ensured that configuration $c(k)$ has received the most recent value known to i, which, as a result of the query phase, is itself a recent value. At this point, the configurations with index $< k$ are no longer needed, and node i removes these configurations from its local *cmap*, setting $cmap(\ell) = \pm$ for all $\ell < k$ (line I–J). Gossip messages may eventually notify other processes that these configurations have been removed.

Finally, a cfg-upgrade-ack(k) event notifies the client that configuration $c(k)$ has been successfully upgraded.

Notice that the algorithm allows a nondeterministic choice of which configuration to upgrade – and therefore which configurations to remove. Therefore it is possible to restrict the algorithm so that it removes only the smallest configuration, upgrading the configurations one at a time. In this case the algorithm progresses exactly as the original RAMBO algorithm. Therefore it is clearly possible, by restricting the nondeterminism appropriately, to implement RAMBO II in such a way as to guarantee equivalent performance as RAMBO. However we will show that by allowing greater flexibility we can achieve equivalent safety results and improved performance.

The new algorithm introduces several difficulties not present in RAMBO. Consider, for example, a nice property guaranteed by the sequential garbage collection algorithm in RAMBO: every configuration is upgraded before it is removed. In RAMBO II, on the other hand, some configurations

[1]In the conference version of the paper, this line was omitted. The removal of this line has no detrimental effect on the algorithm, since the operation then completes in zero time. However for clarity sake it is included.

Operation initiated by $\mathsf{cfg\text{-}upgrade}(k)_i$**:**

Phase 1 :

- Node i communicates with a read-quorum from each configuration being removed in order to determine the largest value/tag pair.
- Node i communicates with a write-quorum from each configuration being removed to notify it of the new, active configuration.

Phase 2 :

- Node i communicates with a write-quorum from the target configuration being upgraded, to notify it of the current largest value/tag pair.

Figure 8: Summary of two phase configuration upgrade operation

never receive up to date information; a configuration may be upgraded at the same instant it is removed.

As a result of this fact, a number of plausible improvements fail. Assume that during an ongoing upgrade operation for configuration $c(k)$ initiated by node i, node i receives a message indicating that configuration $c(k')$ has been removed, for some $k' < k$. In RAMBO II, node i sets $cmap(k') = \pm$, but does not change $upg.cmap$. Consider the following incorrect modification to the configuration management mechanism. When node i receives such a message, it sets $upg.cmap(k')$ to $\pm$. Since the configuration has been removed, it seems plausible that the configuration upgrade operation can safely ignore it, thus completing more quickly. It turns out, however, that this improvement results in a race condition that can lead to data loss. The configuration upgrade operation that removes configuration $c(k')$ might occur concurrently with the operation at node i upgrading configuration $c(k)$. This concurrency might result in data being propagated from configuration $c(k')$ to a configuration $c(k'') : k' < k'' < k$ that has already been processed by the upgrade operation at node i. The data thus propagated might then be lost.

4 Notation and Basic Lemmas

This section is, to a large extent, a restatement of notation and results from the original RAMBO paper [13]. Some of the notation in the proofs has been slightly modified to account for the new configuration management mechanism, and some of the proofs have therefore been updated, but the results are essentially unchanged. Much of this section is taken directly from [13].

4.1 Good Executions

Throughout the rest of this paper, we will talk about "good" executions of the algorithm. In this section, we present a set of environment assumptions that define a "good" execution. In general, the assumptions we will present require well-formed requests: clients follow the protocol to join and to initiate reconfigurations; clients initiate only one operation at a time; clients wait for appropriate acknowledgments before proceeding.

We consider executions of $\mathcal{S}$ (recall that $\mathcal{S}$ is the entire system combining *Reader-Writer*, *Recon* and *Joiner* automata) whose traces satisfy certain assumptions about the environment. We call these *good* executions. In particular, an "invariant" is a statement that is true of all states that are reachable in good executions of $\mathcal{S}$. The environment assumptions are simple "well-formedness" conditions:

- *Well-formedness for Reader-Writer:*
  - For every x and i:
    - No $\mathsf{join}(\mathsf{rambo},*)_{x,i}$, $\mathsf{read}_{x,i}$, $\mathsf{write}(*)_{x,i}$, or $\mathsf{recon}(*,*)_{x,i}$ event is preceded by a fail_i event.
    - At most one $\mathsf{join}(\mathsf{rambo},*)_{x,i}$ event occurs.
    - Any $\mathsf{read}_{x,i}$, $\mathsf{write}(*)_{x,i}$, or $\mathsf{recon}(*,*)_{x,i}$ event is preceded by a $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,i}$ event.
    - Any $\mathsf{read}_{x,i}$, $\mathsf{write}(*)_{x,i}$, or $\mathsf{recon}(*,*)_{x,i}$ event is preceded by an -ack event for any preceding event of any of these kinds.
  - For every x and c, at most one $\mathsf{recon}(*,c)_{x,*}$ event occurs. (This says that configuration identifiers that are proposed in recon events are unique. It does not say that the membership and/or quorum sets are unique—just the identifiers. The same membership and quorum sets may be associated with different configuration identifiers.) Uniqueness of configuration identifiers is achievable using local process identifiers and sequence numbers.
  - For every c, c', x, and i, if a $\mathsf{recon}(c,c')_{x,i}$ event occurs, then it is preceded by:
    - A $\mathsf{report}(c)_{x,i}$ event, and
    - A $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,j}$ event for every $j \in \mathit{members}(c')$.

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

4.2 Notational conventions

In this section, we introduce some definitions and notational conventions, and we add certain history variables to the global state of the system $\mathcal{S}$.

Definitions:

- *update*, a binary function on $C_\pm$, defined by $\mathit{update}(c,c') = \max(c,c')$ if c and c' are comparable (in the augmented partial ordering of $C_\pm$), $\mathit{update}(c,c') = c$ otherwise.

- *extend*, a binary function on $C_\pm$, defined by $\mathit{extend}(c,c') = c'$ if $c = \bot$ and $c' \in C$, and $\mathit{extend}(c,c') = c$ otherwise.

[2]The following properties appear in Section 6, but we repeat them here for completeness.

- *CMap*, the set of *configuration maps*, defined as the set of mappings from $\mathbb{N}$ to $C_{\pm}$. The *update* and *extend* operators are extended element-wise to binary operations on *CMap*.

- *truncate*, a unary function on *CMap*, defined by $truncate(cm)(k) = \perp$ if there exists $\ell \leq k$ such that $cm(\ell) = \perp$, $truncate(cm)(k) = cm(k)$ otherwise. This truncates configuration map *cm* by removing all the configuration identifiers that follow a $\perp$.

- *Truncated*, the subset of *CMap* such that $cm \in Truncated$ if and only if $truncate(cm) = cm$.

- *Usable*, the subset of *CMap* such that $cm \in Usable$ iff the pattern occurring in *cm* consists of a prefix of finitely many $\pm$s, followed by an element of C, followed by an infinite sequence of elements of $C \cup \{\perp\}$ in which all but finitely many elements are $\perp$.

An *operation* is a pair (n, i) consisting of a natural number n and an index $i \in I$. Here, i is the index of the process running the operation, and n is the value of $pnum1_i$ just after the read, write, or cfg-upgrade event of the operation occurs.

We introduce the following history variables:

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

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

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

- For every operation π:

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

- For every read or write operation π:

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

- For every configuration upgrade operation γ for k:

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

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

1. For every read or write operation π:

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

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

4.3 Configuration map invariants

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

Lemma 4.1 *1. If $cm, cm' \in Usable$ then $update(cm, cm') \in Usable$.*

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

3. If $cm, cm' \in Usable$ then $extend(cm, cm') \in Usable$.

4. If $cm \in Usable$ then $truncate(cm) \in Usable$.

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

The next invariant (recall from Section 4.1 that this means a property of all states that arise in good executions of $\mathcal{S}$) describes some properties of $cmap_i$ that hold while *Reader-Writer*$_i$ is conducting a configuration upgrade operation:

Invariant 4.2 *If $upg.phase_i \neq idle$ and $upg.target_i = k$, then:*

1. $\forall \ell : \ell \leq k \Rightarrow cmap(\ell)_i \in C \cup \{\pm\}$.

2. If $k_1 = \min\{\ell : \ell \leq k$ *and* $upg.cmap(\ell) \neq \pm\}$ *then* $k_1 = 0$ *or* $cmap(k_1 - 1)_i = \pm$.

Proof. By the precondition of $\textsf{cfg-upgrade}(k)_i$ and monotonicity of all the changes to $cmap_i$. $\square$

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

Invariant 4.3 *Let cm be a CMap that appears as one of the following:*

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

Then $cm \in Usable$.

In the following proof and elsewhere, we use dot notation to indicate components of a state, for example, $s.cmap_i$ indicates the value of $cmap_i$ in state s.

Proof. By induction on the length of a finite good execution.

Base: Part 1 holds because initially, *in-transit* is empty. Part 2 holds because initially, for every i, $cmap(0)_i = c_0$ and $cmap(k)_i = \perp$; the resulting *CMap* is in *Usable*. Part 3 and Part 5 hold vacuously, because in the initial state, all *op.phase* and *upg.phase* values are *idle*. Part 4 also holds vacuously, because in the initial state, all *query-cmap* and *prop-cmap* variables are undefined.

Inductive step: Let s and s' be the states before and after the new event, respectively. We consider Parts 1–5 one by one.
For Part 1, the interesting case is a $\textsf{send}_i$ event that puts a message containing cm in *in-transit*. The precondition on the $\textsf{send}$ action implies that cm is set to $s.cmap_i$. The inductive hypothesis, Part 2, implies that $s.cmap_i \in Usable$, which suffices.
For Part 2, fix i. The interesting cases are those that may change $cmap_i$, namely, $\textsf{new-config}_i$, $\textsf{recv}_i$ for a gossip (non-join) message, and $\textsf{cfg-upg-prop-fix}_i$. The latter case is the only one modified from the original RAMBO algorithm.

1. $\textsf{new-config}(c, *)_i$.
   By inductive hypothesis, $s.cmap_i \in Usable$. The only change this can make is changing a $\perp$ to c. Then Lemma 4.1, Part 2, implies that $s'.cmap_i \in Usable$.

2. $\textsf{recv}(\langle *, *, cm, *, * \rangle)_i$.
   By inductive hypothesis, $cm \in Usable$ and $s.cmap_i \in Usable$. The step sets $s'.cmap_i$ to $update(s.cmap_i, cm)$. Lemma 4.1, Part 1, then implies that $s'.cmap_i \in Usable$.

3. $\mathsf{cfg\text{-}upg\text{-}prop\text{-}fix}(k)_i$.
   This sets $cmap(\ell)_i$ to $\pm$ for all $\ell < k$. By the definition of this step, $s'.cmap(\ell)_i = \pm$ for $\ell < k$.

   If $s.cmap(k-1)_i = \pm$, then the operation has no effect, and $s'.cmap_i = s.cmap_i \in \mathit{Usable}$. Assume, then, that $s.cmap(k-1)_i \in C \cup \{\bot\}$. This implies, by the inductive hypothesis showing $s.cmap_i \in \mathit{Usable}$, that $s.cmap(\ell)_i \in C \cup \{\bot\}$ for all $\ell \geq k-1$. By Invariant 4.2, we know that $s.cmap(k)_i \in C \cup \{\pm\}$, and therefore $s.cmap(k)_i \in C$. Therefore $s'.cmap(k)_i \in C$ and $s'.cmap(\ell)_i \in C \cup \{\bot\}$ for all $\ell > k$, since the cfg-upg-prop-fix does not change entries in the $cmap$ larger than $k-1$. Further, there are only finitely many entries in $s.cmap_i$ that are in C (by the inductive hypothesis), and so there are still only finitely many entries in $s'.cmap_i$. Therefore, $s'.cmap_i \in \mathit{Usable}$.

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

1. read_i, write_i, or $\mathsf{query\text{-}fix}_i$.
   By inductive hypothesis, $s.cmap_i \in \mathit{Usable}$. The new step sets $s'.op.cmap_i$ to $truncate(s.cmap_i)$; since $s.cmap_i \in \mathit{Usable}$, Lemma 4.1, Part 4, implies that this is also usable.

2. $\mathsf{recv}(\langle *, *, cm, *, * \rangle)_i$.
   This step may alter $op.cmap_i$ only if $s.op.phase \in \{query, prop\}$, and then in only two ways: by setting it either to $extend(s.op.cmap_i, truncate(cm))$ or to $truncate(update(s.cmap_i, cm))$. The inductive hypothesis implies that $s.op.cmap_i$, $cmap_i$, and cm are all in Usable. Lemma 4.1 implies that $truncate$, $extend$, and $update$ all preserve usability. Therefore, $s'.op.cmap_i \in \mathit{Usable}$.

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

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

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

For Part 5, the actions to consider are $\mathsf{cfg\text{-}upgrade}(k)_i$ and $\mathsf{cfg\text{-}upg\text{-}query\text{-}fix}(k)_i$. These set $s'.upg.cmap_i$ to the value of $s.cmap_i$. Since by the inductive hypothesis the latter is usable, so is $s'.upg.cmap_i$.
□

We now strengthen Invariant 4.3 to say more about the form of the *CMaps* that are used for read and write operations:

Invariant 4.4 *Let cm be a CMap that appears as $op.cmap_i$ for some $i \in I$ for which $op.phase_i \neq idle$, or as $query\text{-}cmap(\pi)$ or $prop\text{-}cmap(\pi)$ for any operation π. Then:*

1. *$cm \in Truncated$.*

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

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

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

1. read_i, write_i, or $query\text{-}fix_i$.
   These set $op.cmap_i$ to $truncate(cmap_i)$, which is necessarily in $Truncated$.

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

To see Part 2, note that $cm \in Usable$ by Invariant 4.3. The fact that $cm \in Truncated$ then follows from the definition of $Usable$ and Part 1. □

4.4 Phase guarantees

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

Note that these lemmas treat the case where $j = i$ uniformly with the case where $j \neq i$. This is because, in the *Reader-Writer* algorithm, communication from a location to itself is treated uniformly with communication between two different locations. We first consider the query phase of a configuration-upgrade:

Lemma 4.5 *Suppose that a* $\mathsf{cfg\text{-}upg\text{-}query\text{-}fix}(k)_i$ *event for configuration upgrade operation* γ *occurs in* 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* $\mathsf{cfg\text{-}upgrade}(k)_i$ *event of* γ.

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

3. *m' is received before the* $\mathsf{cfg\text{-}upg\text{-}query\text{-}fix}(k)_i$ *event of* γ.

4. *In any state after j receives m, $cmap(\ell)_j \neq \bot$ for all $\ell \leq k$.*

5. *$tag(\gamma) \geq t$, where t is the value of tag_j in any state before j sends message m'.*

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

For Part 4, the precondition of $\mathsf{cfg\text{-}upgrade}(k)$ implies that, when the $\mathsf{cfg\text{-}upgrade}(k)_i$ event of γ occurs, $cmap(\ell)_i \neq \bot$ for all $\ell \leq k$. Therefore, j sets $cmap(\ell)_j \neq \bot$ for all $\ell \leq k$ when it receives m. Monotonicity of $cmap_j$ ensures that this property persists forever.

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

Next, we consider the propagation phase of a configuration upgrade:

Lemma 4.6 *Suppose that a* $\mathsf{cfg\text{-}upg\text{-}prop\text{-}fix}(k)_i$ *event for a configuration upgrade operation* γ *occurs in* 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* $\mathsf{cfg\text{-}upg\text{-}query\text{-}fix}(k)_i$ *event of* γ*.*

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

3. m' *is received before the* $\mathsf{cfg\text{-}upg\text{-}prop\text{-}fix}(k)_i$ *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'.

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

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

Lemma 4.7 *Suppose that a* $\mathsf{query\text{-}fix}_i$ *event for a read or write operation* π *occurs in* 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'.

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

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

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

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

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

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

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

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

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

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

   By the case assumption, $op.cmap(\ell)_i = \bot$ just before the recv step. Since, by Invariant 4.4, $op.cmap_i \in Truncated$, it follows that $op.cmap(\ell')_i = \bot$ before the recv step.

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

Since this argument holds for all ℓ, $0 \leq \ell \leq k'$, it follows that $cm'(\ell) \neq \bot$ for all $\ell \leq k'$. Since $cm'(\ell) \neq \bot$ for all $\ell \leq k'$, Invariant 4.3 implies that $cm' \in Usable$, which implies by definition of *Usable* that $cm'(\ell) \in C$ for some $\ell \geq k'$. That is, $op.cmap_i(\ell) \in C$ for some $\ell \geq k'$ immediately after the recv step. This implies that $query\text{-}cmap(\pi)(\ell) \in C$ for some $\ell \geq k'$, as needed. □

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

Lemma 4.8 *Suppose that a* prop-fix$_i$ *event for a read or write operation* π *occurs in* 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* prop-phase-start(π) *event.*

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

3. m' *is received before the* prop-fix *event of* π.

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

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'.

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

For Part 5, the proof is analogous to the proof of Part 5 of Lemma 4.7. In fact, it is identical except for the final conclusion, which now says that $prop\text{-}cmap(\pi)(\ell) \in C$ for some $\ell \geq k'$. □

5 Atomic Consistency

This section contains the proof of atomic consistency. The proof is carried out in several stages. First in Section 5.1 we present some lemmas about the new configuration management mechanism, describing the relationship between configuration upgrade operations. Section 5.2 describes the relationship between read/write operations and configuration upgrade operations. Section 5.3 then considers two read or write operations, and culminates in Lemma 5.11, which says that tags are monotonic with respect to non-concurrent read or write operations. Finally, Section 5.4 uses the tags to define a partial order on operations and verifies the four properties required for atomicity.

5.1 Behavior of configuration upgrade

This section presents the key new technical lemmas on which the proof of atomicity is based. Specifically, we present lemmas describing information flow between configuration upgrade operations. These lemmas assert the existence of a sequence of configuration upgrade operations on which we can make certain necessary guarantees. In particular, the key property is that the tags are monotonically increasing with respect to the specific sequence of upgrade operations, guaranteeing that value/tag information is propagated to newer configurations.

The first lemma shows that if all configuration upgrade operations remove two particular configurations together, then those two configuration are always in the same state in all *cmap*s.

Lemma 5.1 *Suppose that $k > 0$, and α is an execution in which no* $\mathsf{cfg\text{-}upg\text{-}prop\text{-}fix}(k)$ *event occurs in α. Suppose that cm is a CMap that appears as one of the following in any state in α:*

1. *The cm component of some message in in-transit.*
2. *$cmap_i$ for any $i \in I$.*

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

Proof. Fix some α and $k > 0$ such that no $\mathsf{cfg\text{-}upg\text{-}prop\text{-}fix}(k)$ event occurs in α. We proceed by induction on the length of a finite prefix of α: for every action in α, if before the action $cm(k-1) = \pm \implies cm(k) = \pm$, then the same implication holds after the action.

Base: For Part 1, the conclusion follows vacuously because initially *in-transit* is empty. For Part 2, the conclusion again follows vacuously because initially $cmap_i(\ell) \neq \pm$ for all i and ℓ.

Inductive step: Let s and s' be the states before and after the new event, respectively. We consider Parts 1 and 2 separately.
For Part 1, the interesting case is a send_i event that puts a message containing cm in *in-transit*. The precondition on the send action implies that cm is set to $s.cmap_i$. The inductive hypothesis, Part 2, implies that if $s.cmap(k-1) = \pm$, then $s.cmap(k) = \pm$. Therefore in state s', the same holds for cm, which has been added to *in-transit*.
For Part 2, fix i. The interesting cases are those that may change $cmap_i$, namely, $\mathsf{new\text{-}config}_i$, recv_i for a gossip message, and $\mathsf{cfg\text{-}upg\text{-}prop\text{-}fix}_i$.

1. $\mathsf{new\text{-}config}(c, *)_i$.
   If $s'.cmap(k-1)_i = \pm$, then $s.cmap(k-1)_i = \pm$, since installing a new configuration does not set any entry to $\pm$. Then by the inductive hypothesis $s.cmap(k)_i = \pm$, which implies that $s'.cmap(k)_i = \pm$, since this action cannot modify an entry that is already $\pm$.

2. $\mathsf{recv}(\langle *, *, cm, *, * \rangle)_i$.
   First, if $cm(0) \neq \pm$, then the message does not cause any entry in $s.cmap$ to be set to $\pm$, and as in Case 1 the desired property still holds. Also, if $s.cmap(0) \neq \pm$, then for all ℓ, $s'.cmap(\ell) = \pm$ if and only if $cm(\ell) = \pm$. By the inductive hypothesis $cm(k-1) = \pm \implies$

$cm(k) = \pm$, so the desired conclusion follows. For the rest of this case, we will assume that $cm(0) = \pm$ and $s.cmap(0) = \pm$.

By Invariant 4.3, $cm \in Usable$. Therefore we can define $k_{msg\text{-}max}$ such that $cm(\ell) = \pm$ for all $\ell \leq k_{msg\text{-}max}$ and $cm(\ell) \neq \pm$ for all $\ell > k_{msg\text{-}max}$. Similarly, we can define k_{max} such that $s.cmap(\ell)_i = \pm$ for all $\ell \leq k_{max}$ and $s.cmap(\ell)_i \neq \pm$ for all $\ell > k_{max}$. Define k'_{max} in the same way for the poststate, s'.

There are two cases. First, assume $k_{max} \geq k_{msg\text{-}max}$. Then $k'_{max} = k_{max}$, by the monotonicity of *CMap*. By our inductive hypothesis $s.cmap(k-1) = \pm \implies s.cmap(k) = \pm$; it follows, then, that if $k-1 \leq k_{max}$ then $k \leq k_{max}$. Therefore if $k-1 \leq k'_{max}$, then $k \leq k'_{max}$. Finally, then, if $s'.cmap(k-1) = \pm$, then $s'.cmap(k) = \pm$.

Assume, then, that $k_{msg\text{-}max} > k_{max}$. Then after the update operation, $k'_{max} = k_{msg\text{-}max}$. By our inductive hypothesis, $cm(k-1) = \pm \implies cm(k) = \pm$; it follows, then, that if $k-1 \leq k_{msg\text{-}max}$, then $k \leq k_{msg\text{-}max}$. Therefore if $k-1 \leq k'_{max}$, then $k \leq k'_{max}$. Finally, then, $s'.cmap(k-1) = \pm$ implies that $s'.cmap(k) = \pm$.

3. cfg-upg-prop-fix$(k')_i$.
   By assumption, $k \neq k'$. If $k < k'$, then this operation sets both $s'.cmap(k-1)_i = \pm$ and $s'.cmap(k)_i = \pm$. If $k > k'$, then this operation has no effect on $cmap(k)_i$ or $cmap(k-1)_i$, and the desired property still holds.

□

The following corollary says that if a cfg-upgrade(k) event for an upgrade operation γ occurs in an execution, then there is some previous configuration upgrade operation γ' (that completes before the upgrade event) where the target of γ' is the configuration with the smallest index removed by γ.

Corollary 5.2 *Let γ be a configuration upgrade operation, initiated by a* cfg-upgrade$(k)_i$ *event in α, and let $k_1 = \min\{removal\text{-}set(\gamma)\}$. That is, k_1 is the smallest element such that $upg\text{-}cmap(\gamma)(k_1) \in C$. Assume $k_1 > 0$. Then a* cfg-upg-prop-fix$(k_1)_j$ *event for some configuration upgrade operation γ' occurs in α for some j such that the* cfg-upg-prop-fix$_j$ *event of γ' precedes the* cfg-upgrade$(k)_i$ *event in α.*

Proof. By the definition of k_1, we know that in the state just after the cfg-upgrade event, $upg.cmap(k_1-1)_i = \pm$ and $upg.cmap(k_1)_i \neq \pm$. Since $upg.cmap_i$ is set by the cfg-upgrade event to $cmap_i$ in the state just prior to the cfg-upgrade event, we know that $cmap(k_1-1)_i = \pm$ and $cmap(k_1)_i \neq \pm$ in the state just prior to the cfg-upgrade event. Lemma 5.1, then, implies that some cfg-upgrade-prop-fix(k_1) event for some operation γ' occurs in α preceding the cfg-upgrade event. □

The next lemma says that for a given configuration upgrade operation γ, there exists a sequence of preceding upgrade operations satisfying certain properties. The lemma begins by assuming that some configuration with index k is removed by the specified upgrade operation. For every configuration with an index smaller than k, we choose a single upgrade operation – that removes that configuration – to add to the sequence. Therefore the constructed sequence may well contain the same configuration upgrade operation multiple times, if the operation has removed multiple configurations. If two elements in the sequence are distinct upgrade operations, then the earlier operation in the sequence completes before the later operation in the sequence is initiated. Also, the target of an upgrade operation in the sequence is removed by the next distinct upgrade operation in the sequence. As a result of these properties, the configuration upgrade process obeys a sequential discipline.

Lemma 5.3 *If a* $\mathsf{cfg\text{-}upgrade}_i$ *event for upgrade operation* γ *occurs in* α *such that* $k \in \textit{removal-set}(\gamma)$*, then there exists a sequence (possibly containing repeated elements) of configuration upgrade operations* $\gamma_0, \gamma_1, \ldots, \gamma_k$ *with the following properties:*

1. $\forall\ s : 0 \leq s \leq k,\ s \in \textit{removal-set}(\gamma_s)$,

2. $\forall\ s : 0 \leq s < k$, *if* $\gamma_s \neq \gamma_{s+1}$*, then the* cfg-upg-prop-fix *event of* γ_s *occurs in* α *and the* cfg-upgrade *event of* γ_{s+1} *occurs in* α*, and the* cfg-upg-prop-fix *event of* γ_s *precedes the* cfg-upgrade *event of* γ_{s+1}*, and*

3. $\forall\ s : 0 \leq s < k$*, if* $\gamma_s \neq \gamma_{s+1}$*, then* $\textit{target}(\gamma_s) \in \textit{removal-set}(\gamma_{s+1})$.

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 following three properties at each step of the induction:

1′. $\forall\ s : \ell \leq s \leq k,\ s \in \textit{removal-set}(\gamma_s)$,

2′. $\forall\ s : \ell \leq s < k$, if $\gamma_s \neq \gamma_{s+1}$, then the cfg-upg-prop-fix event of γ_s occurs in α and the cfg-upgrade event of γ_{s+1} occurs in α, and the cfg-upg-prop-fix event of γ_s precedes the cfg-upgrade event of γ_{s+1}, and

3′. $\forall\ s : \ell \leq s < k$, if $\gamma_s \neq \gamma_{s+1}$, then $\textit{target}(\gamma_s) \in \textit{removal-set}(\gamma_{s+1})$.

To begin the induction, we first examine the base case, where $\ell = k$. Define $\gamma_k = \gamma$. Property 1′ holds by assumption, and Property 2′ and Property 3′ are vacuously true.

For the inductive step, we assume that γ_ℓ has been defined and that properties 1′–3′ hold. 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 properties still hold.

Otherwise, $\ell - 1 \notin \textit{removal-set}(\gamma_\ell)$ and $\ell \in \textit{removal-set}(\gamma_\ell)$, which implies that $\ell = \min\{\textit{removal-set}(\gamma_\ell)\}$ because each configuration upgrade operates on a consecutive sequence of configurations. Then by Corollary 5.2, there occurs in α a configuration upgrade operation, that we label $\gamma_{\ell-1}$, with the following properties: (i) the cfg-upg-prop-fix event of $\gamma_{\ell-1}$ precedes the cfg-upgrade event of γ_ℓ, and (ii) $\textit{target}(\gamma_{\ell-1}) = \min\{k' : k' \in \textit{removal-set}(\gamma_\ell)\}$.

Recall that $\ell = \min\{\textit{removal-set}(\gamma_\ell)\}$. Therefore, by Property (ii) of $\gamma_{\ell-1}$, $\textit{target}(\gamma_{\ell-1}) = \ell$. Since $\textit{removal-set}(\gamma_{\ell-1}) \neq \emptyset$, this implies that $\ell - 1 \in \textit{removal-set}(\gamma_{\ell-1})$, proving Property 1′. Property 2′ follows from Property (i) of $\gamma_{\ell-1}$. Property 3′ follows from Property (ii) of $\gamma_{\ell-1}$. □

The sequential nature of configuration upgrade has a nice consequence for propagation of tags: for any sequence of upgrade operations like that in Lemma 5.3, $\textit{tag}(\gamma_s)$ is nondecreasing in s.

Lemma 5.4 *Let* $\gamma_\ell, \ldots, \gamma_k$ *be a sequence of configuration upgrade operations such that:*

1. $\forall\ s : 0 \leq s \leq k,\ s \in \textit{removal-set}(\gamma_s)$,

2. *$\forall\ s : 0 \leq s < k$, if $\gamma_s \neq \gamma_{s+1}$, then the* cfg-upg-prop-fix *event of γ_s occurs in α and the* cfg-upgrade *event of γ_{s+1} occurs in α, and the* cfg-upg-prop-fix *event of γ_s precedes the* cfg-upgrade *event of γ_{s+1}, and*

3. *$\forall\ s : 0 \leq s < k$, if $\gamma_s \neq \gamma_{s+1}$, then $target(\gamma_s) \in removal\text{-}set(\gamma_{s+1})$.*

Then $\forall\ s : 0 \leq s < k,\ tag(\gamma_s) \leq tag(\gamma_{s+1})$.

Proof. If $\gamma_s = \gamma_{s+1}$, then it is trivially true that $tag(\gamma_s) \leq tag(\gamma_{s+1})$. Therefore assume that $\gamma_s \neq \gamma_{s+1}$; this implies that the cfg-upg-prop-fix event of γ_s precedes the cfg-upgrade event of γ_{s+1}. Let k_2 be the largest element in $removal\text{-}set(\gamma_s)$. We know by assumption that $k_2 + 1 \in removal\text{-}set(\gamma_{s+1})$. Therefore, $W_2(\gamma_s)$, a write-quorum of configuration $c(k_2 + 1)$, has at least one element in common with $R(\gamma_{s+1}, k_2 + 1)$; label this node j. By Lemma 4.6, and the monotonicity of tag_j, after the cfg-upg-prop-fix event of γ_s we know that $tag_j \geq tag(\gamma_s)$. Then by Lemma 4.5 $tag(\gamma_{s+1}) \geq tag_j$. Therefore $tag(\gamma_s) \leq tag(\gamma_{s+1})$. □

Corollary 5.5 *Let $\gamma_\ell, \ldots, \gamma_k$ be a sequence of configuration upgrade operations such that:*

1. *$\forall\ s : 0 \leq s \leq k,\ s \in removal\text{-}set(\gamma_s)$,*

2. *$\forall\ s : 0 \leq s < k$, if $\gamma_s \neq \gamma_{s+1}$, then the* cfg-upg-prop-fix *event of γ_s occurs in α and the* cfg-upgrade *event of γ_{s+1} occurs in α, and the* cfg-upg-prop-fix *event of γ_s precedes the* cfg-upgrade *event of γ_{s+1}, and*

3. *$\forall\ s : 0 \leq s < k$, if $\gamma_s \neq \gamma_{s+1}$, then $target(\gamma_s) \in removal\text{-}set(\gamma_{s+1})$.*

Then $\forall\ s, s' : 0 \leq s \leq s' \leq k,\ tag(\gamma_s) \leq tag(\gamma_{s'})$

Proof. This follows immediately from Lemma 5.4 by induction. □

5.2 Behavior of a read or a write following a configuration upgrade

Now we describe the relationship between an upgrade operation and a following read or write operation. These three lemmas relate the *removal-set* of a preceding configuration upgrade operation with the *query-cmap* of a later read or write operation.

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

Lemma 5.6 *Let π be a read or write operation whose* query-fix *event occurs in α. Let k be the smallest element such that $query\text{-}cmap(\pi)(k) \in C$. Assume $k > 0$. Then there must exist a configuration upgrade operation γ such that $k = target(\gamma)$, and the* cfg-upg-prop-fix *event of γ precedes the* query-phase-start(π) *event.*

Proof. This follows from Lemma 5.1. Let s be the state just before the query-phase-start(π) event. By definition, $query\text{-}cmap(\pi) = s.cmap_i$. Since $s.cmap(k-1)_i = \pm$ and $s.cmap(k)_i \neq \pm$, there must exist such a configuration upgrade operation for k by the contrapositive of Lemma 5.1. □

Second, if some upgrade removing k does complete before the query-phase-start event of a read or write operation, then some configuration with index $\geq k+1$ must be included in the *query-cmap* of a later read or write operation.

Lemma 5.7 *Let γ be a configuration upgrade operation such that $k \in$ removal-set(γ). Let π be a read or write operation whose* query-fix *event occurs in α. Suppose that the* cfg-upg-prop-fix *event of γ precedes the* query-phase-start(π) *event in α.*
Then query-cmap$(\pi)(\ell) \in C$ for some $\ell \geq k+1$.

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

Let $\gamma_0, \ldots, \gamma_k$ be the sequence of upgrade operations whose existence is asserted by Lemma 5.3, where $\gamma_k = \gamma$. Then, by this construction, $k' \in \textit{removal-set}(\gamma_{k'})$, and the cfg-upg-prop-fix event of $\gamma_{k'}$ does not come after the cfg-upg-prop-fix event of γ in α. By assumption, the cfg-upg-prop-fix event of γ precedes the query-phase-start(π) event in α. Therefore the cfg-upg-prop-fix event of $\gamma_{k'}$ precedes the query-phase-start(π) event in α.

Then, since $k' \in \textit{removal-set}(\gamma_{k'})$, write-quorum $W_1(\gamma_{k'}, k')$ is defined. Since *query-cmap*$(k') \in C$), the read-quorum $R(\pi, k')$ is defined. Choose $j \in W_1(\gamma_{k'}, k') \cap R(\pi, k')$. Assume that $k_t = \textit{target}(\gamma_{k'})$. Notice that $k' < k_t$. Then Lemma 4.5 and monotonicity of *cmap* imply that, in the state just prior to the cfg-upg-query-fix event of $\gamma_{k'}$, $\textit{cmap}(\ell)_j \neq \perp$ for all $\ell \leq k_t$. Then Lemma 4.7 implies that *query-cmap*$(\pi)(\ell) \in C$ for some $\ell \geq k_t$. But this contradicts the choice of k'. ☐

The next lemma describes propagation of *tag* information from a configuration upgrade operation to a following read or write operation. For this lemma, we assume that *query-cmap*$(k) \in C$, where k is the target of the upgrade operation,

Lemma 5.8 *Let γ be a configuration upgrade operation. Assume that $k =$ target(γ). Let π be a read or write operation whose* query-fix *event occurs in α. Suppose that the* cfg-upg-prop-fix *event of γ precedes the* query-phase-start(π) *event in execution α. Suppose also that query-cmap$(\pi)(k) \in C$. Then:*

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

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

Lemma 4.6 implies that, in any state after the cfg-upg-prop-fix event for γ, $\textit{tag}_j \geq \textit{tag}(\gamma)$. Since the cfg-upg-prop-fix event of γ precedes the query-phase-start(π) event, we have that $t \geq \textit{tag}(\gamma)$, where t is defined to be the value of $\textit{tag}_j$ just before the query-phase-start(π) event. Then Lemma 4.7 implies that $\textit{tag}(\pi) \geq t$, and if π is a write operation, then $\textit{tag}(\pi) > t$. Combining the inequalities yields both conclusions of the lemma. ☐

5.3 Behavior of sequential reads and writes

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

Lemma 5.9 *Assume π_1 and π_2 are two read or write operations, such that:*

1. *The* prop-fix *event of π_1 occurs in 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 $\min(\{\ell : \textit{prop-cmap}(\pi_1)(\ell) \in C\}) \leq \max(\{\ell : \textit{query-cmap}(\pi_2)(\ell) \in C\})$.

Proof. Suppose for the sake of contradiction that $\min(\{\ell : \textit{prop-cmap}(\pi_1)(\ell) \in C\}) > k$, where k is defined to be $\max(\{\ell : \textit{query-cmap}(\pi_2)(\ell) \in C\})$. Then in particular, $\textit{prop-cmap}(\pi_1)(k) \notin C$. The form of $\textit{prop-cmap}(\pi_1)$, as expressed in Invariant 4.4, implies that $\textit{prop-cmap}(\pi_1)(k) = \pm$.

This implies that some cfg-upg-prop-fix event for some upgrade operation γ such that $k \in \textit{removal-set}(\gamma)$ occurs prior to the prop-fix of π_1, and hence prior to the query-phase-start(π_2) event. Lemma 5.7 then implies that $\textit{query-cmap}(\pi_2)(\ell) \in C$ for some $\ell \geq k+1$. But this contradicts the choice of k. $\square$

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

Lemma 5.10 *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 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 4.8 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 4.7 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. $\square$

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

Lemma 5.11 *Assume π_1 and π_2 are two read or write operations, such that:*

1. *The* prop-fix *of π_1 occurs in 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. $tag(\pi_1) \leq tag(\pi_2)$.
2. *If π_2 is a write then $tag(\pi_1) < tag(\pi_2)$.*

Proof. Let i_1 and i_2 be the indices of the processes that run operations 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 5.10 implies the conclusions of the lemma. So from now on, we assume that no such k exists.

Lemma 5.9 implies that $\min(\{\ell : cm_1(\ell) \in C\}) \leq \max(\{\ell : cm_2(\ell) \in C\})$. Invariant 4.4 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 5.6 implies that there exists a configuration upgrade operation that we will call γ_{s_2-1} such that $s_2 = target(\gamma_{s_2-1})$, and the cfg-upg-prop-fix of γ_{s_2-1} precedes the query-phase-start(π_2) event. Then by Lemma 5.8, $tag(\gamma_{s_2-1}) \leq tag(\pi_2)$, and if π_2 is a write operation then $tag(\gamma_{s_2-1}) < tag(\pi_2)$.

Next we will demonstrate a chain of configuration upgrade operations with non-decreasing tags. Lemma 5.3, in conjunction with the already defined γ_{s_2-1}, implies the existence of a sequence of configuration upgrade operations $\gamma_0, \ldots, \gamma_{s_2-1}$ such that:

1. $\forall\, s : 0 \leq s \leq s_2 - 1,\ s \in \textit{removal-set}(\gamma_s)$,
2. $\forall\, s : 0 \leq s < s_2 - 1$, if $\gamma_s \neq \gamma_{s+1}$, then the cfg-upg-prop-fix event of γ_s precedes the cfg-upgrade event of γ_{s+1} in α,
3. $\forall\, s : 0 \leq s < s_2 - 1$, if $\gamma_s \neq \gamma_{s+1}$, then $target(\gamma_s) \in \textit{removal-set}(\gamma_{s+1})$.

As a special case of Property 1, since $s_1 \leq s_2 - 1$, we know that $s_1 \in \textit{removal-set}(\gamma_{s_1})$. Then Corollary 5.5 implies that $tag(\gamma_{s_1}) \leq tag(\gamma_{s_2-1})$.

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 configuration upgrade 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 4.8 for the propagation phase of γ_{s_1}.

Let message m_2 from the process running γ_{s_1} to j and message m_2' from j to the process running γ_{s_1} be the messages whose existence is asserted in Lemma 4.5 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 $s_t = target(\gamma_{s_1}$. Notice that $s_t > s_1$, since $s_1 \in \textit{removal-set}(\gamma_{s_1})$. Lemma 4.5 implies that in any state after j receives m_2, before j sends m_2', $cmap(k)_j \neq \bot$ 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 4.8 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 4.8 implies that, at the time j sends m'_2, $tag(\pi_1) \leq tag_j$. Then Lemma 4.5 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. □

5.4 Atomicity

In order to prove that all executions of RAMBO II are atomic, we use four sufficient conditions. A memory is said to be *atomic* provided that the following conditions hold for all good executions:

- If all the read and write operations that are invoked complete, then the read and write operations for object x can be partially ordered by an ordering $\prec$, so that:
  1. No operation has infinitely many other operations ordered before it.
  2. The partial order is consistent with the external order of invocations and responses, that is, there do not exist read or write operations π_1 and π_2 such that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$.
  3. All write operations are totally ordered and every read operation is ordered with respect to all the writes.
  4. Every read operation ordered after any writes returns the value of the last write preceding it in the partial order; any read operation ordered before all writes returns the initial value.

This definition is sufficient to guarantee atomicity in terms of the other common definition which is defined in terms of equivalence to a serial memory. (See, for example, Lemma 13.16 in [11].)

Let β be a trace of $\mathcal{S}$, the system that implements RAMBO II (recall that this includes the *Reader-Writer*, *Recon* and *Joiner* automata), 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 $\leq t$ and before all writes with tags $> t$.

Lemma 5.12 *The ordering $\prec$ is well-defined.*

Proof. The key is to show that no two write operations get assigned the same tag. This is obviously true for two writes that are initiated at different locations, because the low-order tiebreaker identifiers are different. For two writes at the same location, Lemma 5.11 implies that the tag of the second is greater than the tag of the first. This suffices. □

Lemma 5.13 *$\prec$ satisfies the four conditions in the definition of atomicity.*

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

1. π_2 is a write operation.

   Since π_1 completes before π_2 starts, Lemma 5.11 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 5.11 implies that $tag(\pi_2) \geq tag(\pi_1)$. On the other hand, the fact that $\pi_2 \prec \pi_1$ implies that $tag(\pi_2) < tag(\pi_1)$. This yields a contradiction.

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

Property 1 follows from Property 2. Properties 3 and 4 are straightforward. □

Now we tie everything together for the proof of Theorem 5.14.

Theorem 5.14 *Let β be a trace of $\mathcal{S}$, the system that implements* RAMBO II. *Then β satisfies the atomicity guarantee.*

Proof. Assume that all read and write operations complete in β. Let α be a good execution of $\mathcal{S}$ whose trace is β. Define the ordering $\prec$ on the read and write operations in β as above, using the execution α. Then Lemma 5.13 says that $\prec$ satisfies the four conditions in the definition of atomicity. Thus, β satisfies the atomicity condition, as needed. □

6 Reconfiguration Service

In this section we present the specification and implementation for the reconfiguration specification. This section is a restatement of Sections 4 and 7 of the RAMBO technical report, and is taken directly from [13]. Our RAMBO implementation for each object x consists of a main *Reader-Writer* algorithm and a reconfiguration service, *Recon*(x); since we are suppressing mention of x, we write this simply as *Recon*. First, in Section 6.1, we present the specification for the *Recon* service, as an external signature and set of traces. In Section 6.2, we present our implementation of *Recon*.

6.1 Reconfiguration Service Specification

The interface for *Recon* appears in Figure 9. The client of *Recon* at location i requests to join the reconfiguration service by performing a $\mathsf{join(recon)}_i$ input action. The service acknowledges this with a corresponding $\mathsf{join\text{-}ack}_i$ output action. The client requests to reconfigure the object using a recon_i input, which is acknowledged with a $\mathsf{recon\text{-}ack}_i$ output action. RAMBO reports a new configuration to the client using a report_i output action. Crashes are modeled using fail actions.

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

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

- *Well-formedness:*

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

Figure 9: *Recon*: External signature

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

The safety guarantees provided by the service are as follows:

- *Well-formedness:* For every i:
  - No $\mathsf{join\text{-}ack}(\mathsf{recon})_i$, $\mathsf{recon\text{-}ack}(*)_i$, $\mathsf{report}(*)_i$, or $\mathsf{new\text{-}config}(*, *)_i$ event is preceded by a fail_i event.
  - Any $\mathsf{join\text{-}ack}(\mathsf{recon})_i$ (resp., $\mathsf{recon\text{-}ack}(c)_i$) event has a preceding $\mathsf{join}(\mathsf{recon})_i$ (resp., recon_i) event with no intervening invocation or response action for x and i.
- *Agreement:* If $\mathsf{new\text{-}config}(c, k)_i$ and $\mathsf{new\text{-}config}(c', k)_j$ both occur, then $c = c'$. (No disagreement arises about what the k^{th} configuration identifier is, for any k.)
- *Validity:* If $\mathsf{new\text{-}config}(c, k)_i$ occurs, then it is preceded by a $\mathsf{recon}(*, c)_{i'}$ for some i' for which a matching $\mathsf{recon\text{-}ack}(nok)_{i'}$ does not occur. (Any configuration identifier that is announced was previously requested by someone who did not receive a negative acknowledgment.)
- *No duplication:* If $\mathsf{new\text{-}config}(c, k)_i$ and $\mathsf{new\text{-}config}(c, k')_{i'}$ both occur, then $k = k'$. (The same configuration identifier cannot be assigned to two different positions in the sequence of configuration identifiers.)

6.2 Reconfiguration Service Implementation

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

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

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

6.3 Consensus services

In this section, we specify the behavior we assume for consensus service $Cons(k, c)$, for a fixed $k \geq 1$ and $c \in C$. This behavior can be achieved using the Paxos consensus algorithm [9], as described formally in [14]. Fix V to be the set of consensus values. (In the implementation of the *Recon* service, V will be instantiated as C.) The external signature of $Cons(k, c)$ is given in Figure 10.

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

Figure 10: $Cons(k, c)$: External signature

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

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

And we define the consensus safety guarantees:

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

We assume that the $Cons(k, c)$ service is implemented using the Paxos algorithm [9], as described formally in [14]. This satisfies the safety guarantees described above, based on the safety assumptions:

Theorem 6.1 *If β is a trace of Paxos that satisfies the safety assumptions of $Cons(k, c)$, then β also satisfies the (well-formedness, agreement, and validity) safety guarantees of $Cons(k, c)$.*

The *Paxos* algorithm also satisfies the following latency result:

Theorem 6.2 *Consider a timed execution α of the Paxos algorithm and a prefix α' of α. Suppose that:*

1. *The underlying system "behaves well" after α', in the sense that timing is "normal" (what is called "regular" in [14])*[3] *and no process failures or message losses occur.*

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

3. *There exist $R \in$ read-quorums(c) and $W \in$ write-quorums(c) such that for all $i \in R \cup W$, no* fail_i *event occurs in α.*

Then for every i that does not fail in α, a $\mathsf{decide}(*)_i$ *event occurs, no later than $9d + \varepsilon$ time after the end of α' ($\varepsilon > 0$).*

6.4 Recon automata

A *Recon*$_i$ process is responsible for initiating consensus executions to help determine new configurations, for telling the local *Reader-Writer*$_i$ process about a newly-determined configuration, and for disseminating information about newly-determined configurations to the members of those configurations. The signature and state of *Recon*$_i$ appear in Figures 11, and the transitions in Figure 12.

Signature:

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

State:

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

Figure 11: *Recon*$_i$: Signature and state

Location i joins the *Recon* service when a *join*(recon) input occurs. *Recon*$_i$ responds with a join-ack.

Recon$_i$ includes a state variable *rec-cmap*, which holds a *CMap*: $\mathit{rec\text{-}cmap}(k) = c$ indicates that i knows that c is the kth configuration identifier. If *Recon*$_i$ has learned that c is the kth configuration identifier, it can convey this to its local *Reader-Writer*$_i$ process using a $\mathsf{new\text{-}config}(c, k)_i$ output action, and it can inform any other *Recon*$_j$ process, $j \in \mathit{members}(c)$, by sending a $\langle \mathsf{config}, c, k\rangle$ message. *Recon*$_i$ learns about new configurations either by receiving a decide input from a *Cons* service, or by receiving a config or init message from another process.

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

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

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

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

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

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

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

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

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

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

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

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

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

Input fail_i
Effect:
  $failed \leftarrow true$

Figure 12: $Recon_i$: Transitions.

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

$Recon_i$ initiates participation in a $Cons(k, c)$ algorithm when its consensus data are prepared. After initiating participation in a consensus algorithm, it sends init messages to inform the other members of c about its initiation of consensus. The other members use this information to prepare to participate in the same consensus algorithm (and also to update their *rec-cmap* if necessary). Thus, there are two ways in which $Recon_i$ can initiate participation in consensus: as a result of a local recon event, or by receiving an init message from another $Recon_j$ process.

When $Recon_i$ receives a $\mathsf{decide}(c')_{k,i}$ directly from $Cons(k, c)$, it records configuration c' in *rec-cmap* It also determines if a response to its local client is necessary (if a local reconfiguration operation is active), and determines the response based on whether the consensus decision is the same as the locally-proposed configuration identifier.

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

Theorem 6.3 *The Recon implementation guarantees well-formedness, agreement, and validity.*

7 Conditional Performance Analysis

In this section we give a conditional latency analysis of the new algorithm, focusing on the improvements realized by the aggressive configuration-upgrade mechanism. We show that the new algorithm allows the system to recover rapidly after a period of unreliable network connectivity or bursty reconfiguration. In particular, we prove that if configurations do not fail too rapidly, then progress is guaranteed. First, in Section 7.1, we present a few general definitions. In Section 7.2 and 7.3, we define the executions being considered, and the environmental assumptions that these executions satisfy. Then in Sections 7.5, 7.6, and 7.7, we prove a series of lemmas that describe how long it takes configuration-upgrade operations to complete. Finally, in Section 7.8 we state the main stabilization theorem, and prove that operations will complete as long as the execution assumptions are met. Throughout this section, we compare the results with those proved in Section 9 of the RAMBO technical report [13].

7.1 Definitions

In this section, we present a few basic definitions. These definitions do not depend on timing, but are needed only for the conditional performance analysis. For these definitions, assume that α is an execution.

First we define what it means for a configuration to be installed: configuration c is *installed* when either of the following holds: (i) $c = c_0$ or (ii) for some $k > 0$, for all non-failed $i \in members(c(k-1))$, a $\mathsf{decide}(c)_{k,i}$ event occurs in α. That is, configuration $c = c(k)$ is installed when every non-failed member of configuration $c(k-1)$ performs a $\mathsf{decide}(c(k))$ event.

Next, we define an event that occurs when a configuration is guaranteed to be ready to be upgraded (though an upgrade operation may occur earlier than this event). We define the

$\mathsf{upgrade\text{-}ready}(k)$ event, for $k > 0$, to be the first event in α after which, $\forall \ell \leq k$, the following hold: (i) configuration $c(\ell)$ is installed, and (ii) $\forall i \in \mathit{members}(c(k-1))$ such that i has not failed at the time of the event, $\mathit{cmap}(\ell)_i \neq \bot$.

7.2 Limiting Nondeterminism

The algorithm, as presented, is highly nondeterministic. Therefore for the purposes of analysis, we restrict our attention to a subset of executions in which automata follow certain timing-related rules. For the rest of this paper we assume a fixed constant $d > 0$. We assume that gossip occurs at fixed intervals of time d, and also that in times of good behavior messages are delivered within time d[4].

1. Each node, $i \in I$, performs a $\mathsf{send}_{i,j}$ for all $j \in \mathit{world}_i$ every time d as measured by the local clock of i.

2. Each node, $i \in I$, performs a $\mathsf{send}_{i,j}$ (an "important" send) whenever any of the following occurs:

   - Just after a $\mathsf{recv}(\mathsf{join})_{j,i}$ event occurs, if $\mathit{status}_i = \mathit{active}$.
   - (Responses for messages) Just after a $\mathsf{recv}(*, *, *, *, \mathit{pns}, *)_{j,i}$ event occurs, if $\mathit{pns} > \mathit{pnum2}(j)_i$ and $\mathit{status}_i = \mathit{active}$.
   - Just after a $\mathsf{new\text{-}config}(c, k)_i$ event occurs if $\mathit{status}_i = \mathsf{active}$ and $j \in \mathit{world}_i$.
   - Just after a $\mathsf{recv}(*, *, *, \mathit{cm}, *, *)_{j,i}$ event occurs, if $\mathit{op.phase}_i \neq \mathsf{idle}$ and for some k, $\mathit{cm}(k) \neq \bot$ and $\mathit{cmap}(k)_i = \bot$.
   - Just after a read_i, write_i, or $\mathsf{query\text{-}fix}_i$ event occurs, if $j \in \mathit{members}(c)$, for some c in the range of $\mathit{op.cmap}_i$.
   - Just after a $\mathsf{cfg\text{-}upgrade}(k)_i$ event occurs for configuration-upgrade γ, if $j \in \mathit{members}(\mathit{cmap}(k')_i)$ for any $k' \in \mathit{removal\text{-}set}(\gamma)$.
   - Just after a $\mathsf{cfg\text{-}upg\text{-}query\text{-}fix}(k)_i$ event occurs for configuration-upgrade γ, if $j \in \mathit{members}(\mathit{cmap}(k')_i)$ where $k' = \mathit{target}(\gamma)$.

3. Locally controlled actions of any automaton in the system that have no effects, other than the important sends described just above, are performed only once.

4. If an action is enabled to occur at node i, and has not yet been performed (and therefore is not restricted by the previous rule), then it occurs immediately, with zero time passing.

7.3 The Behavior of the Environment

Much of the analysis in the original RAMBO algorithm makes guarantees about the latency of requests when "normal behavior" holds. In Section 9 of [13], Lynch and Shvartsman begin to examine how the system behaves in executions that achieve normal behavior *after some point*. Here we adopt a similar model. We first define what it means for an execution to exhibit "normal behavior" from some point onward.

For the rest of the paper, we use the following notation: given some time $t \in \mathbb{R}^{\geq 0}$, $J(t, e, \alpha)$ represents the set of all nodes j such that $\mathsf{join\text{-}ack}_j$ occurs no later than time $t - e - 2d$ in α. (Recall

[4] It seems, perhaps, that we should not be using d to represent both these quantities; however for consistency with the original RAMBO presentation, we continue to use this convention.

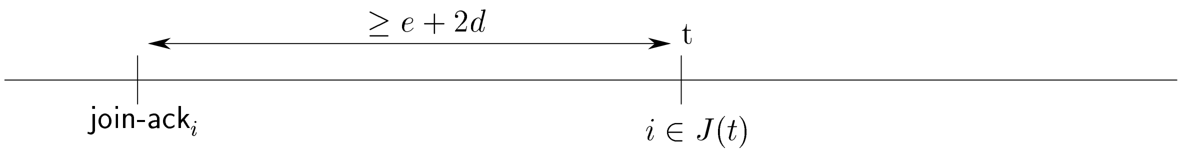

Figure 13: Definition of $J(t)$

that d has been fixed, above.) In most cases, we will use the notation $J(t)$, when e and α are clear from the context.

7.3.1 Normal Timing Behavior from Some Point Onward

Let α be an admissible timed execution, and α' a finite prefix of α. Arbitrary behavior is allowed in α': messages may be lost or delivered late, clocks may run at arbitrary rates, and in general any asynchronous behavior may occur. However we assume that after α', good behavior resumes. We say that α is an α'-*normal* execution if the following assumptions hold:

1. *Initial time:* The join-ack$_{i_0}$ event occurs at time 0, completing the join protocol for node i_0, the node that created the data object.[5]
2. *Regular timing:* The local clocks of all Rambo II automata (i.e., $\mathit{Reader\text{-}Writer}_i$, Recon_i, Joiner_i) at all nodes progress at exactly the rate of real time, after α'.
3. *Reliable message delivery:* No message sent in α after α' is lost.
4. *Message delay bound:* If a message is sent at time t in α and it is delivered, then it is delivered by time $\max(t, \ell time(\alpha')) + d$.

7.3.2 Configuration–Viability

Next we will define *configuration-viability*, which is the key assumption needed to guarantee that read and write operations complete. As in all quorum-based algorithms, liveness depends on all the nodes in some quorums remaining alive. In Rambo II, a node can make progress only if it is able to communicate with the read and write quorums of all extant configurations. We say that a configuration has failed when either: (i) some node in every read-quorum of the configuration has failed, or (ii) some node in every write-quorum of the configuration has failed. If a configuration fails before a new configuration is installed and the old configuration removed, then the system will be effectively crashed: no future read or write request will ever complete. In order to guarantee that operations complete, then, it is necessary for the client using the Rambo II system to initiate appropriate reconfigurations to ensure that quorums remain accessible. The *configuration viability* assumption is a complex property, depending on the behavior of the algorithm, the client initiating appropriate reconfigurations, and on the patterns of node failure and message loss.

We define what it means for an execution to be *(α', e, τ)-configuration-viable*: Let α be an admissible timed execution, and let α' be a finite prefix of α. Let $e, \tau \in \mathbb{R}^{\geq 0}$. Then α is *(α', e, τ)-configuration-viable* if the following holds:
For all i, c, k such that $cmap(k)_i = c$ in some state in α, there exist $R \in \mathit{read\text{-}quorums}(c)$ and $W \in \mathit{write\text{-}quorums}(c)$ such that at least one of the following holds:

[5]This assumption was assumed implicitly in the initial Rambo papers, and was missing from the list of assumptions.

1. No process in $R \cup W$ fails in α.

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 time $\max(\ell time(\alpha') + e, \ell time(\alpha_{install})) + \tau$.

By assuming that an execution is *(α',e,τ)-configuration-viable*, we ensure that the algorithm has at least time τ after a new configuration is installed to clean up obsolete configurations. Also, since all configurations are viable until at least time $e + \tau$ after α', the algorithm has at least time $e + \tau$ after the system stabilizes to clean up obsolete configurations.

7.3.3 Recon-Spacing

While reconfigurations cannot impede a read/write operation, too frequent reconfigurations can slow down a read/write operation by introducing new quorums that must be contacted. In order to bound the time required for a read/write operation, we need to bound the frequency of reconfigurations.

There are two components to Recon-Spacing. Let α be an α'-*normal* execution, and $e \in \mathbb{R}^{\geq 0}$. Then α satisfies:

1. *(α',e)-recon-spacing-1*: if for any $\mathsf{recon}(c,*)_i$ event in α after α' the preceding $\mathsf{report(c)_i}$ event occurs at least time e earlier.

2. *(α',e)-recon-spacing-2*: if for any $\mathsf{recon}(c,*)_i$ event in α after α' there exists a write-quorum $W \in \mathit{write\text{-}quorums}(c)$ such that for all $j \in W$, $\mathsf{report}(c)_j$ precedes the $\mathsf{recon}(c,*)_i$ event in α.

We say that α satisfies *(α',e)-recon-spacing* if it satisfies both *(α',e)-recon-spacing-1* and *(α',e)-recon-spacing-2*.

Notice that, instead of assuming the second part of this requirement, we could instead modify the recon automaton to enforce this ordering: the automaton could collect gossip messages indicating which nodes had performed a $\mathsf{report}(c)$, and delay or abort the next recon if it preceded an appropriate set of report events. We choose to instantiate this as an assumption, rather than as a modification to the automaton for two reasons. First, we prefer to retain compatibility with the original RAMBO analysis. Second, by stating this as an assumption, it is possible that the client using the RAMBO II algorithm might choose to violate the second part of the assumption. As a result, those guarantees that depend on this assumption will not hold; however reconfigurations may be more performed more frequently. Even if the second part of this assumption is violated, safety is still guaranteed: atomicity is maintained, and read and write operations are guaranteed to terminate. However, operations might not terminate rapidly in $8d$, as in Section 7.8.

7.3.4 Join-Connectivity

The hypothesis of *join-connectivity* is designed to ensure that all non-failing joining processes are able to learn about each other. Otherwise, it is possible for the processes to join and fail in such a way that the world-views of the nodes are partitioned into multiple components, with different nodes aware of different, disconnected pieces of the world. It is also important for the latency analysis to bound how long this process takes. If two nodes both complete the join protocol and do not fail, then they should learn about each other within a bounded time. For this reason, we define the notion of *join-connectivity* as follows:

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 $i, j \in J(t, e, \alpha)$, if neither i nor j fails until after $\max(t - 2d, \ell time(\alpha') + e)$, then by time $\max(t - 2d, \ell time(\alpha') + e)$, $i \in world_j$.

This indicates, then, that if two nodes both complete joining by some time t after α', then within time e the two nodes are aware of each other. If two nodes both complete joining by some time t during α', then within time e after α' the two nodes are aware of each other.

Prior results on joining from [13] suggest that in some cases it can be shown that the current simple join protocol in the RAMBO II algorithm provides $(\alpha', d + d\lceil\log(|J|)\rceil)$*-join-connectivity*. However we will not prove - or depend on - this earlier result. Instead we will assume that the system provides *(α',e)-join-connectivity* for some e, and prove our results in this context. We leave it as an open problem to determine the exact value of e; a more complicated and interactive join protocol might well provide better results.

7.3.5 Recon-Readiness

The next assumption we make is related to the problem of reconfiguration by a node that has recently joined. We will assume that every node that is proposed to be a member of a configuration has been a member of the RAMBO II system for a reasonable period of time. This allows us to conclude that everyone is aware of nodes that are part of active configurations.

An α'-*normal* execution α satisfies *(α', e)-recon-readiness* if the following property holds: if for some node i and some configurations c and c', a $\mathsf{recon}(c, c')_i$ event occurs in α at time t, then:

- If $j \in members(c')$, then j performs a join-ack prior to the recon event.
- If the recon event occurs after α', and if $j \in members(c')$, then $j \in J(t, e, \alpha)$.

This prohibits nodes that have just joined the system, but are not yet in anyone's *world* view from forming new configurations. As long as e is not too large, this seems a reasonable requirement.

7.3.6 Upgrade-Readiness

The last assumption we make ensures that a node initiates an upgrade operation only if it has joined sufficiently long ago. This ensures that when a node performs an upgrade, it has relatively up-to-date information.

We say that an α'-*normal* execution α satisfies *(α', e)-upgrade-readiness* if the following property holds: if for some i a $\mathsf{cfg\text{-}upgrade}(*)_i$ event occurs in α after α' at time t, then $i \in J(t)$.

In particular, we suggest that in an implementation of this algorithm, only members of configuration $c(k)$ initiate operations to upgrade configuration $c(k)$. In this case, recon-readiness guarantees upgrade-readiness.

7.3.7 Fixed Parameters

We have already fixed d such that gossip occurs at fixed intervals of time d, and in times of good behaviour messages are delivered with time d. We now fix e as well. Additionally, for the rest of the paper, we fix α and α', and assume that α is an α'-normal execution. We therefore sometimes suppress these parameters, as they are clear from context. For example, we will use the notation $J(t)$ to represent $J(t, e, \alpha)$. When we refer to *join-connectivity*, we mean (α', e)-join-connectivity; *recon-readiness* is used to mean (α', e)-recon-readiness; *upgrade-readiness* is used to mean (α', e)-upgrade-readiness; τ-recon-spacing is used to mean (α', τ)-recon-spacing; τ-configuration-viability is used to mean (α', e, τ)-configuration viability.

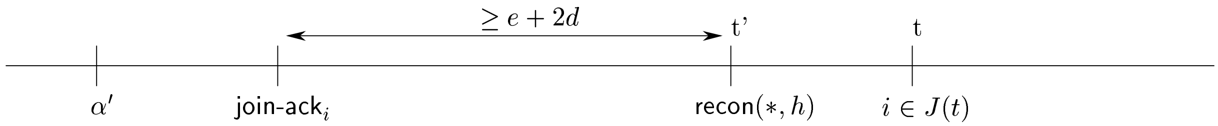

Figure 14: Lemma 7.2, Case 1

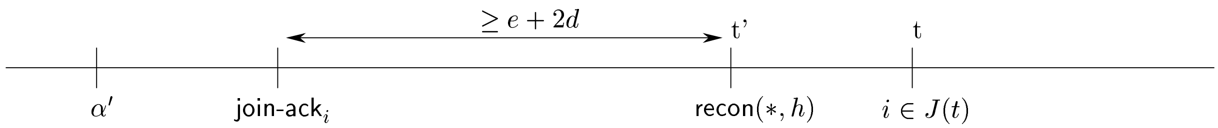

Figure 15: Lemma 7.2, Case 2

7.4 Basic Lemmas

In this section, we prove a few basic lemmas that will be useful in the rest of the paper.

The following two lemmas demonstrate some basic facts about the sets $J(*)$:

Lemma 7.1 *1. If $t \leq t'$, then $J(t) \subseteq J(t')$.*

2. For all t, t', $J(t) \subseteq J(\max(t, t'))$.

Proof. By definition of $J(\cdot)$. $\square$

The following lemma uses the recon-readiness assumption to say something stronger about the joining time of members of a configuration:

Lemma 7.2 *Assume that α is an α'-normal execution satisfying (α', e)-recon-readiness. If h is a configuration proposed at time t' by a* $\mathsf{recon}(*, h)$ *event, $t \geq t'$, and $t \geq \ell time(\alpha') + e + 2d$, then $members(h) \subseteq J(t)$.*

Proof. First, assume that $t' \geq \ell time(\alpha')$. Then the result follows immediately by recon-readiness and Lemma 7.1. Assume, then, that $t' < \ell time(\alpha')$. By recon-readiness, every member of configuration h performs a join-ack by $\ell time(\alpha')$. Therefore, by definition of J, $members(h) \subseteq J(\ell time(\alpha') + e + 2d)$. Since $t \geq \ell time(\alpha') + e + 2d$, Lemma 7.1 implies that $J(\ell time(\alpha') + e + 2d) \subseteq J(t)$. $\square$

The next lemma shows a similar result about upgrade-readiness:

Lemma 7.3 *Assume that α is an α'-normal execution satisfying (α', e)-upgrade-readiness. If a* cfg-upgrade$(*)_i$ *event occurs in α at time t, for some node i, then $i \in J(\max(t, \ell time(\alpha') + e + 2d))$.*

Proof. First, assume that the cfg-upgrade event occurs after α'. Then the lemma follows immediately by the definition of upgrade-readiness and Lemma 7.1. Assume, then, that the cfg-upgrade event occurs in α'. By the precondition of cfg-upgrade, i must perform a join-ack prior to the cfg-upgrade event; otherwise $status_i \neq$ active when the cfg-upgrade occurs, which contradicts the precondition of the cfg-upgrade. Therefore i performs a join-ack$_i$ at latest at time $\ell time(\alpha')$, and therefore $i \in J(\ell time(\alpha') + e + 2d)$, and the lemma again follows by Lemma 7.1. $\square$

7.5 Propagation of Information

In this section, we introduce the notion of information being in the "mainstream". Once a sufficient set of nodes know a particular fact, then, under appropriate assumptions, this fact will never be forgotten by the system as a whole. In particular, we show that this is true about information in the *cmap*: updates to the *cmap* are propagated. Once every non-failed node in $J(t)$ updates its *cmap*, then at any time in the future, at time $t' \geq t + 2d$, every non-failed node in $J(t')$ will be aware of this update.

If *cm* is a CMap and β is a finite prefix of α with $\ell time(\beta) = t \geq e + 2d$, then we say that *cm* is *mainstream* after β provided that the following holds: For every $i \in J(t)$ such that fail_i does not occur in β, $cm \leq \ell state(\beta).cmap_i$.

Further, we define the following notation: given an execution α and a time $t \in \mathbb{R}^{\geq 0}$, we define $\beta(t, \alpha)$ to be the finite prefix of α such that $\ell time(\beta(t, \alpha)) = t$ and every event that occurs at time t occurs in $\beta(t, \alpha)$. As we have already fixed α, for the rest of this paper we use the simpler notation of $\beta(t)$. We then say that a CMap *cm* is mainstream after t if it is mainstream after $\beta(t)$.

The first lemma shows a basic property of mainstream *cmaps*:

Lemma 7.4 *Assume that α is an execution, t is a time, and cm, cm2 are CMaps. If $cm \leq cm2$, and cm2 is mainstream after t, then cm is mainstream after t.*

Proof. Immediate from the definition of mainstream. □

The following lemma shows that a node's *cmap* is monotone:

Lemma 7.5 *Assume that α'' is a finite prefix of execution α, and that α''' is a prefix of α''. Assume that i is a node. Then $\ell state(\alpha''').cmap_i \leq \ell state(\alpha'').cmap_i$.*

Proof. In the algorithm, $cmap_i$ is only modified by the *update* function, and the *update* function is monotone; that is, for all CMaps *new-cmap*, $cmap \leq update(cmap, new\text{-}cmap)$. □

Lemma 7.6 *Assume that α is an execution, and t and t' are times, and that $t \leq t'$. Assume that i is a node, and cm is a CMap.*

1. *If $cm \leq \ell state(\beta(t)).cmap_i$, then $cm \leq \ell state(\beta(t')).cmap_i$.*
2. *$\ell state(\beta(t)).cmap_i \leq \ell state(\beta(t')).cmap_i$.*

Proof. This follows by Lemma 7.5, where $\alpha''' = \beta(t)$ and $\alpha'' = \beta(t')$. □

Next, we demonstrate a particular case when a *cmap* becomes mainstream.

Lemma 7.7 *Let α be an α'-normal execution satisfying* (α',e)-join-connectivity. *Let t be a time such that $t \geq \ell time(\alpha') + e$. If $i \in J(t + 2d)$, and i does not fail in $\beta(t + d)$, then $\ell state(\beta(t)).cmap_i$ is mainstream after $t + 2d$.*

Proof. Let $cm = \ell state(\beta(t)).cmap_i$. To show that *cm* is mainstream after $t + 2d$, we need to show that for all $j \in J(t + 2d)$ such that j does not fail in $\beta(t + 2d)$, $cm \leq \ell state(\beta(t + 2d)).cmap_j$. Fix any such j. By join-connectivity, $j \in world_i$ by time $\max(t, \ell time(\alpha') + e) \leq t$.

By time $t + d$, i sends a gossip message, *msg*, to node j such that $cm \leq msg.cmap_i$. By time $t+2d$, j receives the gossip message and updates $cmap_j$ with $msg.cmap$. By the monotonicity of the *update* function, $msg.cmap \leq update(cmap_j, msg.cmap)$. Therefore $cm \leq \ell state(\beta(t + 2d)).cmap_j$, as required. □

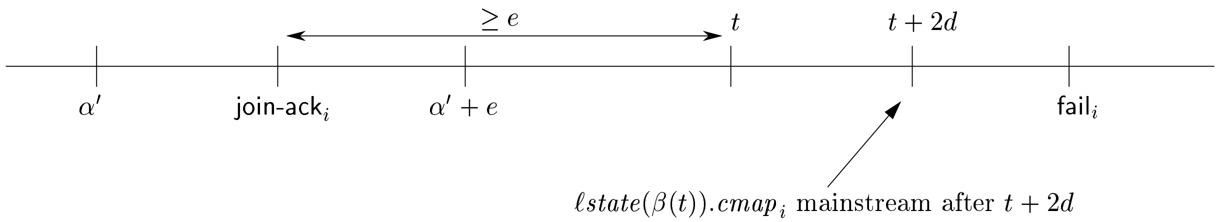

Figure 16: Lemma 7.7

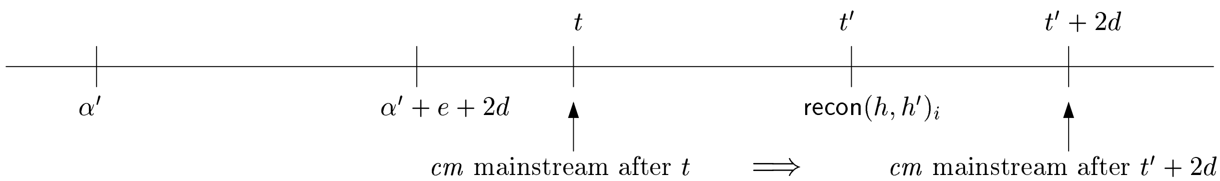

Figure 17: Lemma 7.9

The following lemma shows that if two nodes are both in the set $J(t + 2d)$, then information is propagated from one to the other.

Lemma 7.8 *Let α be an α'-normal execution satisfying* (α',e)-join-connectivity. *Assume that t and t' are times, and $t' - 2d \geq t \geq \ell time(\alpha') + e$. Assume that i and j are nodes, and $i, j \in J(t+2d)$. Also, assume that i does not fail in $\beta(t + 2d)$, and j does not fail in $\beta(t')$.*

If $cm \leq \ell state(\beta(t)).cmap_i$, then $cm \leq \ell state(\beta(t')).cmap_j$.

Proof. By Lemma 7.7, $\ell state(\beta(t)).cmap_i$ is mainstream after $t + 2d$. Notice that $j \in J(t + 2d)$, and therefore, by the definition of mainstream, $\ell state(\beta(t)).cmap_i \leq \ell state(\beta(t+2d)).cmap_j$. Since $t + 2d \leq t'$, by Lemma 7.6, $\ell state(\beta(t + 2d)).cmap_j \leq \ell state(\beta(t')).cmap_j$. Putting the inequalities together, $cm \leq \ell state(\beta(t')).cmap_j$. $\square$

We now show that once a *cmap* is in the mainstream, after $2d$ it will always be in the mainstream. First, Lemma 7.9 considers a special case: it considers only times t' after the system has stabilized, when a recon(h, h') event occurs. Second, Lemma 7.10 handles the case where the *cmap* is in the mainstream at a time in α'. Third, Lemma 7.11 proves the existence of a configuration with some necessary special properties to prove the main theorem. Finally, Lemmas 7.12 and 7.13 prove the general result, as summarized in Lemma 7.14.

First, we define a *successful* recon event as follows: a recon$(*, c)$ event is *successful* if at some time afterwards a decide$(c)_{k,i}$ event occurs for some k and i.

Lemma 7.9 *Let α be an α'-normal execution satisfying: (i)* (α',e)-join-connectivity, *(ii)* (α', e)-recon-readiness, *(iii)* (α', $2d$)-recon-spacing-1, *and (iv)* (α', e, $2d$)-configuration-viability.

Assume that t and t' are times, and that $t \geq \ell time(\alpha') + e + 2d$ and $t' \geq t$. Let h and h' be two configurations, and assume that recon$(h, h')_*$ *occurs at time t', and that this is a successful* recon *event.*

If cm is mainstream after t, then cm is mainstream after $t' + 2d$.

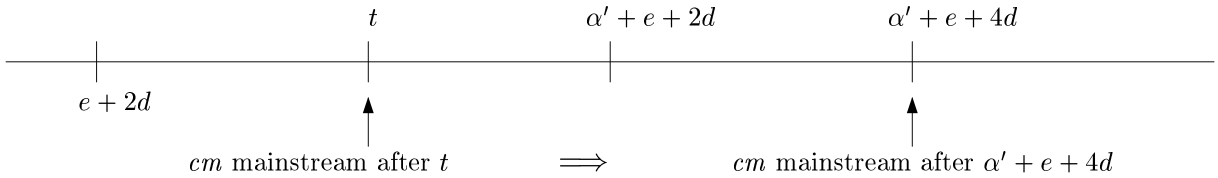

Figure 18: Lemma 7.10

Proof. Fix t and cm such that cm is mainstream after t. We prove the result by induction on the number of successful recon events that occur at or after time t.

As the base case, consider the first successful $\mathsf{recon}(h, h')$ event that occurs in α at a time $t' \geq t$. We need to show that cm is mainstream after $t' + 2d$. Therefore fix some $j' \in J(t' + 2d)$ such that $\mathsf{fail}_{j'}$ does not occur in $\beta(t' + 2d)$. We will show that $cm \leq \ell state(\beta(t' + 2d)).cmap_{j'}$.

Choose some node $j \in members(h)$ such that j does not fail in $\beta(t' + 2d)$; that is, j does not fail until after $t' + 2d$. Configuration-viability ensures that such a node exists. Notice that $j \in J(t)$, by Lemma 7.2. Since cm is mainstream after t, then $cm \leq \ell state(\beta).cmap_j$.

Note that configuration h is proposed prior to time t, since the $\mathsf{recon}(h, h')$ event is the first successful recon event at or after time t. Therefore configuration h is also proposed prior to time t'. By Lemma 7.1, $j \in J(t' + 2d)$. By assumption $j' \in J(t' + 2d)$ and does not fail in $\beta(t' + 2d)$. Therefore, by Lemma 7.8, $cm \leq \ell state(\beta(t' + 2d)).cmap_{j'}$, as needed.

Next we show the inductive step. Inductively assume the following: if $\mathsf{recon}(*, *)$ is one of the first n successful recon events in α that occur at time $t' \geq t$, then cm is mainstream after t'.

Consider the $(n + 1)^{\mathrm{st}}$ successful $\mathsf{recon}(h, h')$ event in α that occurs at or after t. Assume this event occurs at time t'. We need to show that cm is mainstream after $t' + 2d$. Therefore fix some $j' \in J(t' + 2d)$ such that $\mathsf{fail}_{j'}$ does not occur in $\beta(t' + 2d)$. We will show that $cm \leq \ell state(\beta(t' + 2d)).cmap_{j'}$.

Let ρ be the n^{th} successful $\mathsf{recon}(*, h)$ event, and assume that ρ occurs at time t_1. Note that the first argument of the $(n+1)^{\mathrm{st}}$ successful recon event must be the configuration proposed by the n^{th} successful recon event.

$2d$-recon-spacing-1 guarantees that $t' \geq t_1 + 2d$. The inductive hypothesis shows that cm is mainstream after $t_1 + 2d$.

Choose some node $j \in members(h)$ such that no fail_j occurs in $\beta(t'+2d)$. Configuration-viability ensures that such a node exists. By recon-readiness and Lemma 7.1, $j \in J(t' + 2d)$. By assumption $j' \in J(t' + 2d)$ and j' does not fail in $\beta(t' + 2d)$. By Lemma 7.8, $cm \leq \ell state(\beta(t' + 2d)).cmap_{j'}$, as needed. $\square$

The next lemma considers the case where a $cmap$ is mainstream in α' or soon after, and shows that it is mainstream after $\ell time(\alpha') + e + 4d$.

Lemma 7.10 *Let α be an α'-normal execution satisfying (i)* $(\alpha'$,e)-join-connectivity, *(ii)* $(\alpha'$, $e)$-recon-readiness, *(iii)* $(\alpha', 2d)$-recon-spacing-1, *and (iv)* $(\alpha'$, e, $4d)$-configuration-viability.

Assume that t is a time and that $e + 2d \leq t \leq \ell time(\alpha') + e + 2d$. If cm is mainstream after t, then cm is mainstream after $\ell time(\alpha') + e + 4d$.

Proof. Consider configuration c_0. By configuration-viability, there exists a read-quorum, $R \in$ *read-quorums*(c_0), and a write-quorum, $W \in$ *write-quorums*(c_0) such that no node in $R \cup W$ fails by $\ell time(\alpha') + e + 4d$.

Let $t_1 = \ell time(\alpha') + e + 2d$. Consider $i_0 \in R \cup W$; i_0 does not fail by $\ell time(\alpha') + e + 4d$. Since i_0 performs a join-ack at time 0, by the assumption that α is an α'-*normal* execution, and since $t \geq e + 2d$, $i_0 \in J(t)$. Also note that by Lemma 7.6, $i_0 \in J(t_1)$.

Since cm is mainstream after t, $cm \leq \ell state(\beta(t)).cmap_{i_0}$. Therefore, we know by Lemma 7.6 that $cm \leq \ell state(\beta(t_1)).cmap_{i_0}$. By Lemma 7.7, we know that $\ell state(\beta(t_1)).cmap_{i_0}$ is mainstream after $t_1 + 2d$. Therefore by Lemma 7.4, cm is mainstream after $t_1 + 2d$; that is, cm is mainstream after $\ell time(\alpha') + e + 4d$. □

The next lemma shows the existence of a certain configuration, h', with some particular properties. This will be useful in proving Lemma 7.14.

Lemma 7.11 *Let α be an α'-normal execution satisfying: (i)* (α',e)-join-connectivity, *(ii)* (α', e)-recon-readiness, *(iii)* (α', $2d$)-recon-spacing-1, *and (iv)* (α', e, $4d$)-configuration-viability.

Assume that t and t' are times. Assume that $\ell time(\alpha') + e + 2d \leq t \leq t' - 2d$ and $\ell time(\alpha') + e + 6d \leq t'$. Assume that cm is mainstream after t. Then there exists a configuration h, with index k, with the following properties:

1. *$members(h) \subseteq J(t')$.*
2. *For all members i of configuration h that do not fail in $\beta(t')$, $cm \leq \ell state(\beta(t' - 2d)).cmap_i$.*
3. *No successful* recon$(h, *)$ *event occurs in $\beta(t' - 4d)$.*

Proof. There are three different sub-cases to consider.

1. No successful recon event occurs in $\beta(t' - 4d)$:
   Let $h = c_0$. Notice that $members(h) \subseteq J(t)$, since i_0 (the only member of c_0) completes a join-ack at time 0 (by assumption on α), and $t > \ell time(\alpha') + e + 2d$. This, then, implies Property 1 by Lemma 7.1. Since $i_0 \in J(t)$ and cm is mainstream after t, $cm \leq \ell state(\beta(t)).cmap_{i_0}$. Therefore, since $t \leq t' - 2d$, by Lemma 7.6, $cm \leq \ell state(\beta(t' - 2d)).cmap_{i_0}$, as required for Property 2. Property 3 holds trivially.

2. A successful recon event occurs in $\beta(t' - 4d)$ after time t:
   Consider the last successful recon event in α that occurs in $\beta(t' - 4d)$; let h be the configuration identifier appearing as the second argument in this recon event. Assume that this recon event occurs at time t_{rec}. Note that $t < t_{rec} \leq t' - 4d$. Therefore (since $t' \geq \ell time(\alpha') + e + 6d$ and $t' \geq t_{rec}$) by Lemma 7.2, $members(h) \subseteq J(t')$, as required for Property 1. Since $t_{rec} > t$, Lemma 7.9 shows that cm is mainstream after $t_{rec} + 2d$. Recall that $t_{rec} + 2d \leq t' - 2d$. By the mainstream property, for every member, i, of configuration h that does not fail in $\beta(t' - 2d)$, $cm \leq \ell state(\beta(t_{rec} + 2d)).cmap_i$; therefore, for each of these members, i, by Lemma 7.6, $cm \leq \ell state(\beta(t' - 2d)).cmap_i$, as required for Property 2. Property 3 holds by the selection of the last successful recon event in $\beta(t' - 4d)$.

3. Neither Case 1 nor Case 2 holds, that is, a successful recon event occurs in $\beta(t' - 4d)$, but no such recon event occurs after time t:
   Consider the last successful recon event in α that occurs in $\beta(t' - 4d)$; let h be the configuration identifier appearing as the second argument in this recon event. Assume that this recon event occurs at time t_{rec}. Notice, then, that $t_{rec} \leq t$. (Otherwise, Case 2 would hold.) Since $t \geq \ell time(\alpha') + e + 2d$, then by Lemma 7.2, $members(h) \subseteq J(t)$. By Lemma 7.6, then, $members(h) \subseteq J(t')$, which implies Property 1. Since cm is mainstream after t (and

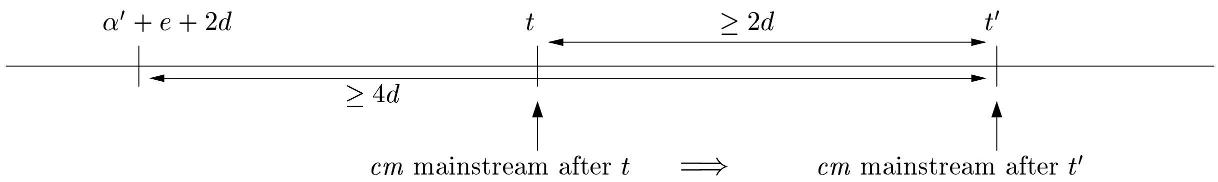

Figure 19: Lemma 7.12

members$(h) \subseteq J(t)$), for all $j \in$ *members*(h) such that no fail_j event occurs in $\beta(t)$, $cm \leq \ell state(\beta(t)).cmap_j$. Since $t \leq t' - 2d$, by Lemma 7.6, for all j such that no fail_i event occurs by time $t' - 2d$, $cm \leq \ell state(\beta(t' - 2d)).cmap_j$, as required for Property 2. Property 3 holds by the selection of the last successful recon event that occurs in $\beta(t' - 4d)$.

□

Finally we prove the main lemma of this section, showing that if a *cmap* is mainstream at time t, then the *cmap* is also mainstream at times $t' \geq t + 2d$. There are two cases to consider: (i) $t \geq \ell time(\alpha') + e + 2d$, and (ii) $t < \ell time(\alpha') + e + 2d$. Lemma 7.12 shows the first case, Lemma 7.13 shows the second case, and Lemma 7.14 presents the overall conclusion.

Lemma 7.12 *Let α be an α'-normal execution satisfying (i)* $(\alpha'$,e)-join-connectivity, *(ii)* $(\alpha',\ e)$-recon-readiness, *(iii)* $(\alpha', 2d)$-recon-spacing-1, *and (iv)* $(\alpha',\ e,\ 4d)$-configuration-viability.

Assume that t and t' are times. Assume that $e + 2d \leq t \leq t' - 2d$ and $\ell time(\alpha') + e + 6d \leq t'$. Additionally assume that $t \geq \ell time(\alpha') + e + 2d$. If cm is a mainstream CMap after t, then cm is mainstream after t'.

Proof. By assumption, $t \geq \ell time(\alpha') + e + 2d$. Lemma 7.11 shows that there exists a configuration, h, with index k with the following three properties:

1. *members*$(h) \subseteq J(t')$.
2. For all members i of configuration h that do not fail in $\beta(t')$, $cm \leq \ell state(\beta(t' - 2d)).cmap_i$.
3. No successful $\mathsf{recon}(h, *)$ event occurs in $\beta(t' - 4d)$.

Configuration-viability guarantees that some node of configuration h does not fail until after the next configuration is installed. No successful $\mathsf{recon}(h, *)$ event occurs in $\beta(t' - 4d)$, by Property 3. Therefore some node, $j \in$ *members*(h) does not fail in $\beta(t')$ (and therefore does not fail in $\beta(t' - d)$), by $4d$-configuration-viability. By Property 1 of h, node $j \in J(t')$. Therefore, by Lemma 7.7, $\ell state(\beta(t' - 2d)).cmap_j$ is mainstream after t'.

Further, we know by Property 2 that $cm \leq \ell state(\beta(t' - 2d)).cmap_j$. Therefore by Lemma 7.4, cm is mainstream after t'. □

The following lemma considers the case where $t < \ell time(\alpha') + e + 2d$:

Lemma 7.13 *Let α be an α'-normal execution satisfying (i)* $(\alpha'$,e)-join-connectivity, *(ii)* $(\alpha',\ e)$-recon-readiness, *(iii)* $(\alpha', 2d)$-recon-spacing-1, *and (iv)* $(\alpha',\ e,\ 4d)$-configuration-viability.

Assume that t and t' are times. Assume that $e + 2d \leq t \leq t' - 2d$ and $\ell time(\alpha') + e + 6d \leq t'$. Additionally, assume that $t < \ell time(\alpha') + e + 2d$. If cm is a mainstream CMap after t, then cm is mainstream after t'.

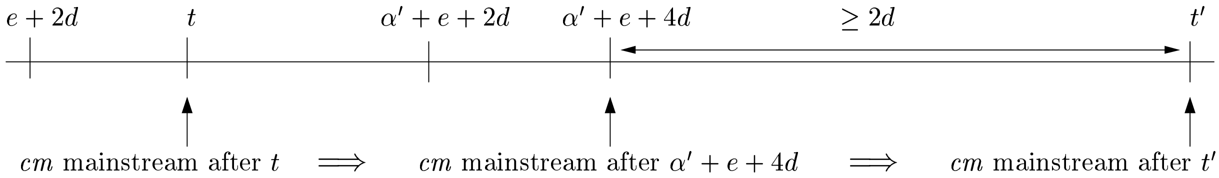

Figure 20: Lemma 7.13

Proof. By assumption, $t < \ell time(\alpha') + e + 2d$. Let $t_1 = \ell time(\alpha') + e + 2d$. By Lemma 7.10, cm is mainstream after $t_1 + 2d$. By assumption, $t_1 + 2d \leq t' - 2d$, and $\ell time(\alpha') + e + 2d \leq t_1 + 2d$. By Lemma 7.12, however, we know that since cm is mainstream after $t_1 + 2d$, then cm is mainstream after t'. $\square$

The following lemma combines the previous two lemmas into a single conclusion. This lemma is the main result of this section, and is used throughout the rest of the proof.

Lemma 7.14 *Let α be an α'-normal execution satisfying (i)* $(\alpha'$,e)-join-connectivity, *(ii)* $(\alpha'$, $e)$-recon-readiness, *(iii)* $(\alpha', 2d)$-recon-spacing-1, *and (iv)* $(\alpha'$, e, $4d)$-configuration-viability.

Assume that t and t' are times. Assume that $e + 2d \leq t \leq t' - 2d$ and $\ell time(\alpha') + e + 6d \leq t'$. If cm is a mainstream CMap after t, then cm is mainstream after t'.

Proof. By Lemmas 7.12 and 7.13. $\square$

7.6 Upgrade-Ready Viability

In this section, we show the relationship between a configuration being upgrade-ready, and a configuration being viable. In particular, we prove that if an execution α is *(α',e,22d)-configuration-viable*, then configuration $c(k)$ is viable until at least $15d$ after the upgrade-ready$(c(k+1))$ event.

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

Lemma 7.15 *Let α be an α'-normal execution satisfying: (i)* $(\alpha'$,e)-join-connectivity, *(ii)* $(\alpha'$, $e)$-recon-readiness, *(iii)* $(\alpha'$, e, $4d)$-configuration-viability.

Assume that $t \in R^{\geq 0}$ is a time, and configuration $c(k)$ is installed at time t. Then there exists a CMap, cm, such that $cm(k) \neq \perp$, and cm is mainstream after $\max(t, \ell time(\alpha') + e) + 2d$.

Proof. We first find a node $j \in members(c(k-1))$ such that $j \in J(\max(t, \ell time(\alpha') + e) + 2d)$ and j does not fail in $\beta(\max(t, \ell time(\alpha') + e) + d)$. Configuration-viability guarantees that there exists a read-quorum $R \in read\text{-}quorums(c(k-1))$ and a prefix α'' of α such that $c(k)$ is installed in α and no node in R fails by $\max(\ell time(\alpha''), \ell time(\alpha') + e) + 4d$. Since configuration $c(k)$ is installed at time t, we know that $t \leq \ell time(\alpha'')$, and therefore no node in R fails by $\max(t, \ell time(\alpha') + e) + 4d$. Therefore no node in R fails in $\beta(\max(t, \ell time(\alpha') + e) + d)$. Choose some node $j \in R$.

Assume that configuration $c(k-1)$ is proposed at time t_{rec}. We next apply Lemma 7.2 where $h = c(k-1)$, $t' = t_{rec}$, and $t = \max(t, \ell time(\alpha') + e) + 2d$:

- $\max(t, \ell time(\alpha') + e) + 2d \geq t_{rec}$: $c(k-1)$ is proposed at $t_{rec} \leq t$, since $c(k-1)$ must be proposed prior to configuration $c(k-1)$ being installed, which must occur prior to configuration $c(k)$ being installed; $t \leq \max(t, \ell time(\alpha') + e) + 2d$.

- $\max(t, \ell time(\alpha') + e) + 2d \geq \ell time(\alpha') + e + 2d$: Immediate.

We therefore conclude that $members(c(k-1)) \subseteq J(\max(t, \ell time(\alpha') + e) + 2d)$. Therefore we have shown that $j \in members(c(k-1))$, $j \in J(\max(t, \ell time(\alpha') + e) + 2d)$, and j does not fail in $\beta(\max(t, \ell time(\alpha') + e) + d)$.

Since configuration $c(k)$ is installed at time t and $j \in members(c(k-1))$, $\ell state(\beta(t)).cmap(k)_j \neq \bot$, by the definition of a configuration being installed, and therefore (by Lemma 7.6) $\ell state(\beta(\max(t, \ell time(\alpha') + e))).cmap(k)_j \neq \bot$. We let $cm = \ell state(\beta(\max(t, \ell time(\alpha') + e))).cmap(k)_j$; $cm(k) \neq \bot$, as required.

We next apply Lemma 7.7, where $t = \max(t, \ell time(\alpha') + e)$ and $i = j$:

- $\max(t, \ell time(\alpha') + e) \geq \ell time(\alpha') + e$: Immediate.
- $j \in J(\max(t, \ell time(\alpha') + e) + 2d)$: Shown above.
- j does not fail in $\beta(\max(t, \ell time(\alpha') + e) + d)$: Shown above.

We therefore conclude that $\ell state(\beta(\max(t, \ell time(\alpha') + e))).cmap_i$ is mainstream after $\max(t, \ell time(\alpha') + e) + 2d$, that is, cm is mainstream after $\max(t, \ell time(\alpha') + e) + 2d$. $\square$

The next lemma shows that soon after smaller configurations are installed, a configuration is upgrade-ready.

Lemma 7.16 *Let α be an α'-normal execution satisfying: (i)* (α',e)-join-connectivity*, (ii)* (α', e)-recon-readiness*, (iii)* (α', $2d$)-recon-spacing-1, *and (iv)* (α', e, $4d$)-configuration-viability*.*

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 upgrade-ready(k) *occurs in* $\beta(\max(t, \ell time(\alpha') + e) + 6d)$.

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

We first show that for all $\ell \leq k$, there exists a CMap, cm_ℓ such that $cm_\ell(\ell) \neq \bot$ and cm_ℓ is mainstream after $\max(t, \ell time(\alpha') + e) + 6d$. Fix some $\ell \leq k$.

Lemma 7.15, where $t = t_\ell$ and $k = \ell$, shows that there exists a CMap, cm_ℓ, such that $cm_\ell(\ell) \neq \bot$ and cm_ℓ is mainstream after time $\max(t_\ell, \ell time(\alpha') + e) + 2d$.

We next apply Lemma 7.14, where $t = \max(t_\ell, \ell time(\alpha') + e) + 2d$ and $t' = \max(t, \ell time(\alpha') + e) + 6d$:

- $\max(t_\ell, \ell time(\alpha') + e) + 2d \geq e + 2d$: Immediate.
- $\max(t_\ell, \ell time(\alpha') + e) + 2d \leq \max(t, \ell time(\alpha') + e) + 6d - 2d$: We know that $t_\ell \leq t$, and $\ell time(\alpha') + e + 2d \leq \ell time(\alpha') + e + 4d$.
- $\max(t, \ell time(\alpha') + e) + 6d \geq \ell time(\alpha') + e + 6d$: Immediate.
- cm_ℓ is mainstream after $\max(t_\ell, \ell time(\alpha') + e) + 2d$: Shown above.

We therefore conclude that cm_ℓ is mainstream after $\max(t, \ell time(\alpha') + e) + 6d$. We have thus shown that for all $\ell \leq k$, there exists a CMap, cm_ℓ such that $cm_\ell(\ell) \neq \bot$ and cm_ℓ is mainstream after $\max(t, \ell time(\alpha') + e) + 6d$.

Recall that upgrade-ready(k) is designated as the first event after which (i) all configurations with index $\leq k$ have been installed, and (ii) for all $\ell < k$, for all members of configuration $c(k-1)$

that do not fail prior to the upgrade event, $cmap(\ell) \neq \perp$. The first component occurs by time t, and therefore by time $\max(t, \ell time(\alpha') + e) + 6d$, by assumption.

We therefore need to show the second part. Fix some node $j \in members(c(k-1))$ such that j does not fail in $\beta(\max(t, \ell time(\alpha') + e) + 6d)$. Fix some $\ell < k$. We apply Lemma 7.2, where $h = c(k-1)$, $t = \max(t, \ell time(\alpha') + e) + 6d$, and t' is the time at which configuration $c(k-1)$ is proposed:

- $\max(t, \ell time(\alpha') + e) + 6d$ is $\geq$ the time at which configuration $c(k-1)$ is proposed: $c(k-1)$ is proposed prior to time t_{k-1} (the time at which configuration $c(k-1)$ is installed), which is $\leq$ time $t \leq \max(t, \ell time(\alpha') + e) + 6d$.

- $\max(t, \ell time(\alpha') + e) + 6d \geq \ell time(\alpha') + e + 2d$: Immediate.

We therefore conclude that $members(c(k-1)) \subseteq J(\max(t, \ell time(\alpha') + e) + 6d)$, and therefore $j \in J(\max(t, \ell time(\alpha') + e) + 6d)$.

We know from above that cm_ℓ is mainstream after $\max(t, \ell time(\alpha') + e) + 6d$, which implies, by the definition of being mainstream, that $cm_\ell \leq \ell state(\beta(\max(t, \ell time(\alpha') + e) + 6d)).cmap(\ell)_j$. This in turn implies that $\ell state(\beta(\max(t, \ell time(\alpha') + e) + 6d)).cmap(\ell)_j \neq \perp$, as required. Therefore upgrade-ready(k) occurs in $\beta(\max(t, \ell time(\alpha') + e) + 6d)$. $\square$

The next lemma directly relates the time when all quorums of configuration $c(k-1)$ fail to the time at which upgrade-ready(k) occurs.

Lemma 7.17 *Let α be an α'-normal execution satisfying: (i)* $(\alpha'$,e)-join-connectivity, *(ii)* (α', e)-recon-readiness, *(iii)* $(\alpha', 2d)$-recon-spacing-1, *and (iv)* $(\alpha', e, 22d)$-configuration-viability.

Let c be a configuration with index k, and assume that the upgrade-ready(k) *event occurs at time t. Then there exists a read-quorum, R, and a write-quorum, W, of configuration $c(k-1)$ such that no node in $R \cup W$ fails in $\beta(\max(t, \ell time(\alpha') + e) + 16d)$.*

Proof. Let α'' be the shortest prefix of α such that every configuration with index $\leq k$ is installed in α. Let $t' = \ell time(\alpha'')$. Notice that for all $\ell \leq k$, configuration $c(\ell)$ is installed in $\beta(t')$.

Lemma 7.16, where $t = t'$ and c and k are as defined above, shows that the upgrade-ready(k) event occurs in $\beta(\max(t', \ell time(\alpha') + e) + 6d)$, that is, $t \leq \max(t', \ell time(\alpha') + e) + 6d$.

Configuration-viability guarantees that there exists a read-quorum, R, and a write-quorum, W, of configuration $c(k-1)$ such that either (1) no process in $R \cup W$ fails in α, or (2) there exists a finite prefix, $\alpha_{install}$ of α such that for all $\ell \leq k$, configuration $c(\ell)$ is installed in $\alpha_{install}$ and no process in $R \cup W$ fails in α by time $\max(\ell time(\alpha_{install}), \ell time(\alpha') + e) + 22d$. In the former case, we are done. We now consider the second case. Since α'' is the shortest prefix of α such that every configuration with index $\leq k$ is installed, we know that α'' is a prefix of $\alpha_{install}$, and therefore $t' = \ell time(\alpha'') \leq \ell time(\alpha_{install})$. Therefore we know that there exists a read-quorum, $R \in read\text{-}quorums(c(k-1))$, and a write-quorum, $W \in write\text{-}quorums(c(k-1))$, such that no node in $R \cup W$ fails by time $\max(t', \ell time(\alpha') + e) + 22d$.

Then, $\max(t, \ell time(\alpha')+e)+16d \leq \max(t', \ell time(\alpha')+e)+22d$, and as a result, no node in $R \cup W$ fails by time $\max(t, \ell time(\alpha') + e) + 16d$. That is, no node in $R \cup W$ fails in $\beta(\max(t, \ell time(\alpha') + e) + 16d)$. $\square$

The final lemma shows that if no upgrade-ready(k) occurs in α, then configuration $c(k-1)$ is always viable.

Lemma 7.18 *Let α be an α'-normal execution satisfying: (i)* (α',e)-join-connectivity, *(ii)* (α', e)-recon-readiness, *(iii)* (α', $2d$)-recon-spacing-1, *and (iv)* (α', e, $4d$)-configuration-viability.

Let c be a configuration with index k, and assume that no upgrade-ready($k+1$) *event occurs in α. Then there exists a read-quorum, $R \in$ read-quorums(c), and a write-quorum, $W \in$ write-quorums(c), 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. By the definition of configuration-viability, then, there exists a read-quorum, $R \in$ *read-quorums*(c), and a write-quorum, $W \in$ *write-quorums*(c), such that no node in $R \cup W$ fails in α.

Assume, instead, that for every $\ell \leq k+1$, configuration $c(\ell)$ is installed in α. Then by Lemma 7.16, an upgrade-ready($k+1$) event occurs in α, contradicting the hypothesis. □

7.7 Configuration-Upgrade Latency Results

In this section we show that configuration-upgrade operations terminate rapidly, and that any obsolete configuration is rapidly removed. In particular, these results hold in executions that include periods of bad behavior. The configuration-upgrade mechanism in RAMBO does not make these guarantees. The original RAMBO latency analysis required the assumption of (α', ∞)-*configuration-viability*[6] for the entire execution. This is an unrealistic assumption in a long-lived dynamic system. As a result of the new configuration-upgrade mechanism, we need to assume only bounded configuration-viability to ensure liveness.

First we state a lemma about configuration-upgrade after the system stabilizes and good behavior resumes.

Lemma 7.19 *Let α be an α'-normal execution. Let $t \in R^{\geq 0}$ be a time. Let i be a node that does not fail until after* $\max(t, \ell time(\alpha') + d) + 4d$.

Assume a cfg-upgrade$(k)_i$ *event occurs in α at time t. Additionally, assume that for every configuration $c(\ell)$ such that $upg.cmap(\ell)_i \in C$, 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 $t+3d$.*

Then a cfg-upgrade-ack$(k)_i$ *event occurs no later than $t+4d$.*

Proof. There are two cases to consider.

Case 1: $t > \ell time(\alpha')$. At time t, node i begins the configuration-upgrade, with phase-number $p_1 = upg.pnum_i$. By triggered gossip, node i immediately sends out messages to every node in $world_i$. Therefore for every configuration $c(\ell)$ such that $upg.cmap(\ell)_i \in C$, every node $j \in R_\ell \cup W_\ell$ receives a message by time $t+d$.

By triggered gossip, then, each of these nodes sends a response with phase-number p_1. Each response is received by time $t+2d$, at which point a cfg-upg-query-fix$(k)_i$ event occurs. Node i then chooses a new phase-number, p_2, and sets $upg.pnum_i = p_2$.

Immediately, by triggered gossip node i sends out messages to every process in $world_i$, including every node in $R_\ell \cup W_\ell$, for every configuration $c(\ell)$ such that $upg.cmap(\ell)_i \in C$. Again, a response is sent by time $t+3d$, and node i receives a response from each with phase-number p_2 by time $t+4d$. Immediately, then, a cfg-upg-query-fix(k) event occurs. This is followed by a cfg-upgrade-ack(k), proving our claim.

[6]Although we have not formally defined (α', ∞)-*configuration-viability* here, one can understand it to mean (α', e)-*configuration-viability* for arbitrarily large e.

Case 2: $t \leq \ell time(\alpha')$. At time t, node i begins the configuration-upgrade, with phase-number $p_1 = upg.pnum_i$. By occasional gossip, i sends out messages to every node in $world_i$. Therefore for every configuration $c(\ell)$ such that $upg.cmap(\ell)_i \in C$, every node $j \in R_\ell \cup W_\ell$ receives a message by time $\max(t, \ell time(\alpha') + d) + d$.

By triggered gossip, then, each of these nodes sends a response with phase-number p_1. Each response is received by time $\max(t, \ell time(\alpha') + d) + 2d$, at which point a cfg-upg-query-fix$(k)_i$ event occurs. Node i then chooses a new phase-number, p_2, and sets $upg.pnum_i = p_2$.

Immediately, by triggered gossip node i sends out messages to every process in $world_i$, including every node in $R_\ell \cup W_\ell$, for every configuration $c(\ell)$ such that $upg.cmap(\ell)_i \in C$. Again, a response is sent by time $\max(t, \ell time(\alpha') + d) + 3d$, and node i receives a response from each with phase-number p_2 by time $\max(t, \ell time(\alpha')) + 4d$. Immediately, then, a cfg-upg-query-fix(k) event occurs. This is followed by a cfg-upgrade-ack(k), proving our claim.

□

Next, we provide a conditional guarantee that a configuration is viable: if for some time t every earlier cfg-upgrade operation completes rapidly within $4d$, then every configuration that is extant at time t will remain viable until $t + 3d$.

We do this in four steps. First, Lemma 7.20 demonstrates that a node with certain good properties exists. Second, Lemma 7.21 shows that this certain node with good properties will begin an upgrade operation, in certain situations. Third, Lemma 7.22 shows that soon after a configuration is upgrade-ready(k), some node completes an upgrade operation on configuration $c(k)$. Finally, Lemma 7.23 uses these preliminary lemmas to show that under certain conditions, configurations remain viable sufficiently long.

Lemma 7.20 *Let α be an α'-normal execution satisfying (i)* $(\alpha'$, e)-join-connectivity*, (ii)*$(\alpha'$, $e)$-recon-readiness*, (iii)* $(\alpha'$, $e)$-upgrade-readiness*, (iv)* $(\alpha', 2d)$-recon-spacing-1*, (v)* $(\alpha'$, e, $22d)$-configuration-viability*.*

Assume that an upgrade-ready(k_2) *event occurs at time t for some configuration c_2 and assume that $k_2 \geq 1$. Let $k_1 = k_2 - 1$, and $c_1 = c(k_1)$. Then there exists a node i such that the following hold:*

1. *i is a member of configuration c_1,*
2. *i does not fail in $\beta(\max(t, \ell time(\alpha') + e + d) + 10d)$,*
3. *$i \in J(\max(t, \ell time(\alpha') + e + d) + 8d)$,*
4. *$i \in J(\max(t, \ell time(\alpha') + e + 2d))$,*
5. *i performs a* join-ack *prior to the* upgrade-ready(k_2) *event in α.*

Proof. Lemma 7.17, applied with $c = c_2$, $k = k_2$, and t as defined above, implies that there exists a read-quorum, R, of configuration c_1 such that no member of R fails in $\beta(\max(t, \ell time(\alpha') + e) + 16d)$. Then we know that no member of R fails in $\beta(\max(t, \ell time(\alpha') + e + d) + 14d)$. We therefore choose a node $i \in R \subseteq members(c_1)$. We know that i does not fail in $\beta(\max(t, \ell time(\alpha') + e + d) + 10d)$. This i satisfies Parts 1 and 2.

Let t_{c_1} be the time at which configuration c_1 is proposed. Notice that $\max(t, \ell time(\alpha') + e + 2d) \geq t_{c_1}$, because t, the time of the upgrade-ready(k_2), cannot be smaller than t_{c_1}, the time at which configuration c_1 is proposed (since an $\mathsf{upgrade\text{-}ready}(k_2)$ event cannot occur until after a $\mathsf{recon}(c_1, c_2)$ event, which cannot occur until after a $\mathsf{recon}(*, c_1)$ event). Therefore, Lemma 7.2, applied where $h = c_1$, $t' = t_{c_1}$, and $t = \max(t, \ell time(\alpha') + e + 2d)$, guarantees that $members(c_1) \subseteq J(\max(t, \ell time(\alpha')+e+2d))$. Since $i \in members(c_1)$, we know that $i \in J(\max(t, \ell time(\alpha')+e+2d))$, satisfying Part 4.

Since $\max(t, \ell time(\alpha') + e + 2d) \leq \max(t, \ell time(\alpha') + e + d) + 10d$ (since $\ell time(\alpha') + e + 2d \leq \ell time(\alpha') + e + 10d$), Lemma 7.1, applied where $t = \max(t, \ell time(\alpha') + e + 2d)$ and $t' = \max(t, \ell time(\alpha') + e + d) + 10d$, implies that $J(\max(t, \ell time(\alpha') + e + 2d)) \subseteq J(\max(t, \ell time(\alpha') + e + d) + 10d)$, and thus $i \in J(\max(t, \ell time(\alpha') + e + d) + 10d)$, satisfying Part 3.

Finally, notice that recon-readiness requires that i performs a $\mathsf{join\text{-}ack}$ prior to the $\mathsf{recon}(*, c_1)$ event, and therefore prior to the $\mathsf{cfg\text{-}upgrade}(k_2)$ event. This satisfies Part 5. □

The next lemma claims that when a configuration is upgrade-ready, and a node with certain properties (as in Lemma 7.20) exists, then either the configuration is removed or an upgrade operation begins.

Lemma 7.21 *Let α be an α'-normal execution satisfying (i) $(\alpha'$, e)-join-connectivity, (ii)$(\alpha'$, e)-recon-readiness, (iii) $(\alpha'$, e)-upgrade-readiness, (iv) $(\alpha', 2d)$-recon-spacing-1, (v) $(\alpha'$, e, $22d)$-configuration-viability.*

Assume $\mathsf{upgrade\text{-}ready}(k_2)$ *occurs at time t and $k_2 \geq 1$. Let $k_1 = k_2 - 1$ and $c_1 = c(k - 1)$. Further, assume that node i has the following properties:*

1. *i is a member of configuration c_1,*
2. *i does not fail in $\beta(\max(t, \ell time(\alpha') + e + d) + 10d)$,*
3. *$i \in J(\max(t, \ell time(\alpha') + e + d) + 8d)$,*
4. *$i \in J(\max(t, \ell time(\alpha') + e + 2d))$,*
5. *i performs a* $\mathsf{join\text{-}ack}$ *prior to the* $\mathsf{upgrade\text{-}ready}(k_2)$ *event.*

Let t' be a time such that $t \leq t' < \max(t, \ell time(\alpha') + e + d) + 13d$. Let α'' be a prefix of α such that:

1. *$t' = \ell time(\alpha'')$,*
2. *an* $\mathsf{upgrade\text{-}ready}(k_2)$ *event is in α'',*
3. *$\ell state(\alpha'').upg.phase_i = \mathsf{idle}$.*

Then either:

1. *$\ell state(\beta(t')).cmap(k_1)_i = \pm$, or*
2. *i performs a* $\mathsf{cfg\text{-}upgrade}(k')_i$ *at time t', for some $k' \geq k_2$.*

Proof. If $\ell state(\alpha'').cmap(k_1)_i = \pm$, then the conclusion holds, since α'' is a prefix of $\beta(t')$: by Lemma 7.6, $\ell state(\beta(t')).cmap(k_1)_i = \pm$. Assume, then, that $\ell state(\alpha'').cmap(k_1)_i \neq \pm$. We examine in turn the preconditions for $\mathsf{cfg\text{-}upgrade}(k')_i$ just after α'' (from Figure 7):

1. $\neg\ell state(\alpha'').failed_i$: By Part 2 of the assumption on i, we know that i does not fail in $\beta(\max(t, \ell time(\alpha') + e + d) + 10d)$. However, $t' < \max(t, \ell time(\alpha') + e + d) + 10d$, and thus i does not fail in $\beta(t')$. Since α'' is a prefix of $\beta(t')$, i does not fail in α''.

2. $\ell state(\alpha'').status_i$ = active: By Part 5 of the assumption on i we know that i performs a join-ack prior to the upgrade-ready(k_2) event.

3. $\ell state(\alpha'').upg.phase_i$ = idle: By assumption, this holds.

4. $\forall \ell \in \mathbb{N}, \ell \leq k_2 : \ell state(\alpha'').cmap(\ell)_i \neq \bot$: It suffices to show that by the point in the execution at which the upgrade-ready(k_2) event occurs, node i has already learned of configuration c_2 and all configurations with smaller indices. Let α''' be the prefix of α ending in the upgrade-ready(k_2) event. Part (ii) of the definition of the upgrade-ready(k_2) event guarantees that: for all $\ell \leq k_2$, for all $j \in members(c_1)$ that do not fail in α''', $\ell state(\alpha''').cmap(\ell)_j \neq \bot$. Notice that by Part 1 of the assumption about i, $i \in members(c_1)$ and that by Part 2 of the assumption about i, i does not fail in α''', since $\ell time(\alpha''') = t \leq \max(t, \ell time(\alpha') + e + d)$. Therefore we can conclude by part (ii) that for all $\ell \leq k_2$, $\ell state(\alpha''').cmap(\ell)_i \neq \bot$. Since α''' is a prefix of α'' (by assumption that upgrade-ready(k_2) is included in α''), by Lemma 7.5 we know that for all $\ell \leq k_2$, $\ell state(\alpha'').cmap(\ell)_i \neq \bot$, as desired.

5. $\ell state(\alpha'').cmap(k_2)_i \in C$: By assumption, $\ell state(\alpha'').cmap(k_1)_i \neq \pm$. Invariant 4.3 then implies that $\ell state(\alpha'').cmap(k_2)_i \neq \pm$, since $k_1 < k_2$. Part 4, above, shows that $\ell state(\alpha'').cmap(k_2)_i \neq \bot$, thus implying the desired result.

6. $\ell state(\alpha'').cmap(k_1)_i \in C$: By assumption, $\ell state(\alpha'').cmap(k_1)_i \neq \pm$. Part 4, above, shows that $\ell state(\alpha'').cmap(k_1)_i \neq \bot$, since $k_1 \leq k_2$, thus implying the desired result.

Since enabled events occur in zero time (by assumption), either the event becomes disabled, in which case $\ell state(\beta(t')).cmap(k_1)_i = \pm$, satisfying Part 1 of the conclusion, or at time $t' = \ell time(\alpha'')$ a cfg-upgrade event for some configuration c with index $k' \geq k_2$ occurs, satisfying Part 2 of the conclusion. □

The next lemma conditionally guarantees that soon after a new configuration is upgrade-ready, the old configuration is removed.

Lemma 7.22 *Let α be an α'-normal execution satisfying (i)* (α', e)-join-connectivity, *(ii)*(α', e)-recon-readiness, *(iii)* (α', e)-upgrade-readiness, *(iv)* (α', 2d)-recon-spacing-1, *(v)* (α', e, 22d)-configuration-viability.

Assume that $t \in R^{\geq 0}$ is a time such that $t > \ell time(\alpha') + e + 14d$. Assume that c_1 is a configuration, and for some finite prefix α'' of α, where $t = \ell time(\alpha'')$, for some node $i \in J(t)$ that does not fail in α'', for some index k_1, $\ell state(\alpha'').cmap(k_1)_i = c_1$.

Also, we assume the Upgrades-Complete Hypothesis: for every cfg-upgrade$(*)_j$ *event that occurs in α at some time $t_{upg} < t$ at some node $j \in J(\max(t_{upg}, \ell time(\alpha')+e+2d))$ where j does not fail in $\beta(\max(t_{upg}, \ell time(\alpha')+e+d)+4d)$, a matching* cfg-upg-ack$(*)_j$ *occurs by time* $\max(t_{upg}, \ell time(\alpha')+e+d)+4d$.

Assume that an upgrade-ready(k_1+1) *event occurs at time $t' < t - 13d$. Let $k_2 = k_1 + 1$ and $c_2 = c(k_2)$. Then for some node $i' \in J(\max(t', \ell time(\alpha') + e + d) + 8d)$ that does not fail in $\beta(\max(t', \ell time(\alpha') + e + d) + 10d)$, $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap(k_1)_{i'} = \pm$.*

Proof. We first identify a node, i', that is suitable. Then we show that i' completes an upgrade operation in the alotted time.

We apply Lemma 7.20, where $t = t'$, and therefore conclude that there exists a node i' with the following five properties:

1. i' is a member of configuration c_1,
2. i' does not fail in $\beta(\max(t', \ell time(\alpha') + e + d) + 10d)$,
3. $i' \in J(\max(t', \ell time(\alpha') + e + d) + 8d)$,
4. $i' \in J(\max(t', \ell time(\alpha') + e + 2d))$,
5. i' performs a join-ack prior to the upgrade-ready(k_2) event.

Notice that Part 2 and Part 3 satisfy the first two requirements for i' in the conclusion of this lemma. It remains to show that i' marks configuration c_1 as $\pm$ at the appropriate point.

We consider what happens at time $\max(t', \ell time(\alpha') + e + d)$. Let α''' be the prefix of α that is the longer of the following two prefixes: (i) $\beta(\ell time(\alpha') + e + d)$, or (ii) the shortest prefix of α that includes the cfg-upgrade(k_2) event. Notice that $\ell time(\alpha''') = \max(t', \ell time(\alpha') + e + d)$, and that the cfg-upgrade(k_2) event is in α'''.

If $\ell state(\alpha''')).cmap(k_1)_{i'} = \pm$, then the claim is immediate: Lemma 7.5 implies that $\ell state(\alpha''').cmap_{i'} \leq \ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap_{i'}$, since $\ell time(\alpha''') = \max(t', \ell time(\alpha') + e + d) < \max(t', \ell time(\alpha') + e + d) + 8d$. Therefore, if $\ell state(\alpha''').cmap(k_1)_{i'} = \pm$, then $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap(k_1)_{i'} = \pm$.

We thus assume that $\ell state(\alpha''').cmap(k_1)_{i'} \neq \pm$, and consider what happens at time $\max(t', \ell time(\alpha') + e + d)$. There are now two cases to consider:

1. $\ell state(\alpha''').upg.phase_{i'} =$ idle or
2. $\ell state(\alpha''').upg.phase_{i'} \neq$ idle.

Case 1: Assume that $\ell state(\alpha''').upg.phase_{i'} =$ idle. We apply Lemma 7.21, where $t = t'$, $t' = \max(t', \ell time(\alpha') + e + d)$, $\alpha'' = \alpha'''$, and i' is as chosen above:

- $t' \leq \max(t', \ell time(\alpha') + e + d) < \max(t', \ell time(\alpha') + e + d) + 13d$: immediate,
- i' satisfies the criteria, by the properties of i' above,
- $\ell time(\alpha''') = \max(t', \ell time(\alpha') + e + d)$ and upgrade-ready(k_2) occurs in α''': by the way in which α'' was chosen,
- $\ell state(\alpha''').upg.phase_{i'} =$ idle: by the case assumption.

From this lemma, we conclude that either:

1. $\ell state(\beta(\max(t', \ell time(\alpha') + e + d))).cmap(k_1)_{i'} = \pm$, or
2. i' performs a cfg-upgrade$(k')_{i'}$ at time $\max(t', \ell time(\alpha') + e + d)$, for some $k' \geq k_2$.

In the first case, where $\ell state(\beta(\max(t', \ell time(\alpha') + e + d))).cmap(k_1)_{i'} = \pm$, we are done: Lemma 7.6 implies that $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap(k_1)_{i'} = \pm$. Consider the second case, that is, i' performs a cfg-upgrade$(k')_{i'}$ at time $\max(t', \ell time(\alpha') + e + d)$, for some $k' \geq k_2$.

We then apply the Upgrades-Complete Hypothesis, where $j = i'$ and $t_{upg} = t'$; notice that:

- $i' \in J(\max(t', \ell time(\alpha') + e + 2d))$: by $4^t h$ property of i',
- i' does not fail in $\beta(\max(t', \ell time(\alpha') + e + d) + 4d)$: by Part 2 of the way in which i' was chosen, and

- $\max(t', \ell time(\alpha') + e + d) < t$: $t' + 13d < t$, by assumption, and $\ell time(\alpha') + e + 14d < t$, by assumption, and therefore $\max(t', \ell time(\alpha') + e + d) + 13d < t$.

Therefore, by the Upgrades-Complete Hypothesis we conclude that a $\mathsf{cfg\text{-}upg\text{-}ack}(k')_{i'}$ occurs by time $\max(t', \ell time(\alpha') + e + d) + 4d$. Since $k' \geq k_2$, then by the precondition of a $\mathsf{cfg\text{-}upg\text{-}ack}$ operation we know that $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 4d).cmap(k_1)_{i'} = \pm$. Lemma 7.6 implies that $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d).cmap(k_1)_{i'} = \pm$, as desired.

Case 2: Assume that $\ell state(\alpha''').upg.phase_{i'} \neq \mathsf{idle}$. For this to occur, a $\mathsf{cfg\text{-}upgrade}(k')_{i'}$ event must occur prior to the $\mathsf{upgrade\text{-}ready}(k_2)$ event in α with no matching $\mathsf{cfg\text{-}upg\text{-}ack}(k')_{i'}$ event prior to the $\mathsf{upgrade\text{-}ready}(k_2)$ event, where $k' = \ell state(\alpha'').upg.target_{i'}$. Otherwise, if there were no ongoing upgrade operation, i' would be idle. Let t_1 be the time at which this earlier $\mathsf{cfg\text{-}upgrade}(k')_{i'}$ operation occurs.

We can then apply the Upgrades-Complete Hypothesis, where $j = i'$ and $t_{upg} = t_1$; notice that:

- $i' \in J(\max(t_1, \ell time(\alpha') + e + 2d))$: Lemma 7.3, applied where $t = t_1$ and $i = i'$, shows that $i' \in J(\max(t_1, \ell time(\alpha') + e + 2d))$.
- i' does not fail in $\beta(\max(t_1, \ell time(\alpha') + e + d) + 4d)$: By Part 2 of the way in which i' was chosen, i' does not fail in $\beta(\max(t', \ell time(\alpha') + e + d) + 10d)$. Notice that $t_1 \leq \max(t', \ell time(\alpha') + e + d)$, since the earlier upgrade event occurs in α''' prior to the $\mathsf{upgrade\text{-}ready}(k_2)$ event. Therefore i' does not fail in $\beta(\max(t_1, \ell time(\alpha') + e + d) + 4d)$.
- $\max(t_1, \ell time(\alpha') + e + d) < t$: Again, notice that $\max(t_1, \ell time(\alpha') + e + d) \leq \max(t', \ell time(\alpha') + e + d)$, since $t_1 \leq t'$. Also, $t' + 13d < t$, by assumption, and $\ell time(\alpha') + e + 14d < t$, by assumption. Therefore, $\max(t', \ell time(\alpha') + e + d) < t$, implying that $\max(t_1, \ell time(\alpha') + e + d) < t$.

We can then conclude that a $\mathsf{cfg\text{-}upgrade\text{-}ack}(k')_{i'}$ occurs in α by time $\max(t_1, \ell time(\alpha') + e + d) + 4d \leq \max(t', \ell time(\alpha') + e + d) + 4d$. If $k' \geq k_2$, then by the precondition of the $\mathsf{cfg\text{-}upgrade\text{-}ack}(k')$ action, i' marks $cmap(k_1) = \pm$, and we are done.

Otherwise, we apply Lemma 7.21 to show that another $\mathsf{cfg\text{-}upgrade}$ operation begins: let t_2 be the time at which the $\mathsf{cfg\text{-}upgrade\text{-}ack}(k')_{i'}$ occurs and α_2 be the prefix of α ending in the $\mathsf{cfg\text{-}upgrade\text{-}ack}(k')_{i'}$ event. Notice that:

- $t' \leq \max(t_2, \ell time(\alpha') + e + d)$: By the way in which the $\mathsf{cfg\text{-}upgrade}(k')$ was chosen, it has to complete no earlier than t'.
- $\max(t_2, \ell time(\alpha') + e + d) < \max(t', \ell time(\alpha') + e + d) + 13d$: Above, we showed that that $\mathsf{cfg\text{-}upgrade\text{-}ack}(k')_{i'}$ occurs by $\max(t', \ell time(\alpha') + e + d) + 4d$, that is, $t_2 \leq \max(t_1, \ell time(\alpha') + e + d) + 4d \leq \max(t', \ell time(\alpha') + e + d) + 4d$, since $t_1 \leq t'$. Therefore, $t_2 < \max(t', \ell time(\alpha') + e + d) + 13d$. Also, $\ell time(\alpha') + e + d < \ell time(\alpha') + e + 14d$.

Then we apply Lemma 7.21 with $t = t'$, $t' = \max(t_2, \ell time(\alpha') + e + d)$, $\alpha'' = \alpha_2$, and i' as chosen above:

- $t' \leq \max(t_2, \ell time(\alpha') + e + d) < \max(t', \ell time(\alpha') + e + d) + 13d$: as shown above,
- i' satisfies the criteria, by the properties of i' above,
- $\ell time(\alpha_2) = \max(t_2, \ell time(\alpha') + e + d)$ and $\mathsf{upgrade\text{-}ready}(k_2)$ occurs in α'': by the way in which α_2 was chosen and the fact that the $\mathsf{cfg\text{-}upgrade\text{-}ack}(k')_{i'}$ must come after the $\mathsf{upgrade\text{-}ready}(k_2)$ event,

- $\ell state(\alpha_2).upg.phase_{i'} = \mathsf{idle}$: by the effect of the $\mathsf{cfg\text{-}upg\text{-}ack}(k')_{i'}$ event that is the last event in α'''.

We then conclude that either:

1. $\ell state(\beta(\max(t_2, \ell time(\alpha') + e + d))).cmap(k_1)_{i'} = \pm$, or
2. i' performs a $\mathsf{cfg\text{-}upgrade}(k'')_{i'}$ at time $\max(t_2, \ell time(\alpha') + e + d)$, for some $k'' \geq k_2$.

Again, if the first case holds, we are done: since $t_2 \leq \max(t', \ell time(\alpha') + e + d) + 8d$, Lemma 7.6 implies that $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap(k_1)_{i'} = \pm$. Therefore, we can assume that the second part holds, and i' performs a $\mathsf{cfg\text{-}upgrade}(k'')_{i'}$ at time $\max(t_2, \ell time(\alpha') + e + d)$, for some $k'' \geq k_2$.

Once more, we apply the Upgrades-Complete Hypothesis, where $j = i'$ and $t_{upg} = t_2$; notice that:

- $i' \in J(\max(t_2, \ell time(\alpha') + e + 2d))$: Recall that $i' \in J(\max(t_1, \ell time(\alpha') + e + 2d))$, above. Since $\max(t_1, \ell time(\alpha') + e + 2d) \leq \max(t_2, \ell time(\alpha') + e + 2d)$ (i.e., the upgrade begins before it completes), by Lemma 7.1, where $t = \max(t_1, \ell time(\alpha') + e + 2d)$ and $t' = \max(t_2, \ell time(\alpha') + e + 2d)$, $J(\max(t_1, \ell time(\alpha') + e + 2d)) \subseteq J(\max(t_2, \ell time(\alpha') + e + 2d))$, implying that $i' \in J(\max(t_2, \ell time(\alpha') + e + 2d))$.
- i' does not fail in $\beta(\max(t_2, \ell time(\alpha') + e + d) + 4d)$: By Part 2 of the way in which i' was chosen, i' does not fail in $\beta(\max(t', \ell time(\alpha') + e + d) + 10d)$. Notice that $t_2 \leq \max(t', \ell time(\alpha') + e + d) + 4d$, as shown above. Therefore $\max(t_2, \ell time(\alpha') + e + d) + 4d \leq \max(t', \ell time(\alpha') + e + d) + 8d$, and as a result i' does not fail in $\beta(\max(t_2, \ell time(\alpha') + e + d) + 4d)$.
- $\max(t_2, \ell time(\alpha') + e + d) < t$: Again, notice that $\max(t_2, \ell time(\alpha') + e + d) \leq \max(t', \ell time(\alpha') + e + d) + 4d$. Also, $t' + 13d < t$, by assumption, and $\ell time(\alpha') + e + d + 13d < t$, by assumption. Therefore, $\max(t', \ell time(\alpha') + e + d) + 13d < t$. Therefore, $\max(t_2, \ell time(\alpha') + e + d) \leq \max(t', \ell time(\alpha') + e + d) + 4d < t - 9d$, as desired.

We can then conclude that a $\mathsf{cfg\text{-}upgrade\text{-}ack}(k'')_{i'}$ occurs in α by time $\max(t_2, \ell time(\alpha') + e + d) + 4d \leq \max(t', \ell time(\alpha') + e + d) + 8d$. Since $k'' \geq k_2$, then by the precondition of the $\mathsf{cfg\text{-}upgrade\text{-}ack}(k')$ action, i' marks $cmap(k_1) = \pm$, and Lemma 7.6 implies that $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap(k_1)_{i'} = \pm$.

□

In the next lemma, we provide a conditional guarantee that a configuration remains viable.

Lemma 7.23 *Let α be an α'-normal execution satisfying (i)* $(\alpha'$, e)-join-connectivity*, (ii)*$(\alpha'$, $e)$-recon-readiness*, (iii)* $(\alpha'$, $e)$-upgrade-readiness*, (iv)* $(\alpha', 2d)$-recon-spacing-1*, (v)* $(\alpha'$, e, $22d)$-configuration-viability*.*

Assume that $t \in R^{\geq 0}$ is a time such that $t > \ell time(\alpha') + e + 14d$. Assume that c_1 is a configuration, and for some finite prefix α'' of α, where $t = \ell time(\alpha'')$, for some node $i \in J(\max(t, \ell time(\alpha') + e + 2d))$ that does not fail in α'', for some index k_1, $\ell state(\alpha'').cmap(k_1)_i = c_1$.

Also we assume the Upgrades-Complete Hypothesis: for all $\mathsf{cfg\text{-}upgrade}(*)_j$ *events that occur in α at some time $t_{upg} < t$ at some node $j \in J(\max(t_{upg}, \ell time(\alpha') + e + 2d))$ where j does not fail in $\beta(\max(t_{upg}, \ell time(\alpha') + e + d) + 4d$, a matching* $\mathsf{cfg\text{-}upg\text{-}ack}(*)_j$ *occurs by time* $\max(t_{upg}, \ell time(\alpha') + e + d) + 4d$.

Then there exists a read-quorum, $R \in$ read-quorums(c_1), and a write-quorum, $W \in$ write-quorums(c_1), such that no node in $R \cup W$ fails in $\beta(t + 3d)$.

Proof. Let $k_2 = k_1 + 1$, and let $c_2 = c(k_2)$. First, consider the case where no upgrade-ready(k_2) event occurs in α. We apply Lemma 7.18, where $c = c_1$ and $k = k_1$; this implies, then, that there exists a read-quorum, $R \in$ *read-quorums*(c_1), and a write-quorum, $W \in$ *write-quorums*(c_1), such that no node in $R \cup W$ fails in α.

Next, consider the case where an upgrade-ready(k_2) event occurs in α. Let t' be the time at which the upgrade-ready(k_2) event occurs. We claim that upgrade-ready(k_2) occurs no earlier than $t - 13d$. That is, $t' + 13d \geq t$.

Assume, in contradiction, that $t' + 13d < t$. We now apply Lemma 7.22 to conclude that there exists a node $i' \in J(\max(t', \ell time(\alpha') + e + d) + 8d)$ that does not fail in $\beta(\max(t', \ell time(\alpha') + e + d) + 10d)$ such that $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap(k_1)_{i'} = \pm$.

We now show that the information about configuration c_1's removal is propagated from node i' to node i. That is, we show the following:

Claim: $\ell state(\alpha'').cmap(k_1)_i = \pm$.

Proof of claim: We do this in three steps. First, we show that $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap_{i'}$ is mainstream after $\max(t', \ell time(\alpha') + e + d) + 10d$. Second, we show that $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap_{i'}$ is mainstream after $t - d$. Third, we conclude that $\ell state(\alpha'').cmap(k_1)_i = \pm$.

Step 1: We already know that $i' \in J(\max(t', \ell time(\alpha') + e + d) + 8d)$, and does not fail in $\beta(\max(t', \ell time(\alpha') + e + d) + 10d)$. We then apply Lemma 7.7, where $t = \max(t', \ell time(\alpha') + e + d) + 8d$, and $i = i'$:

- $\max(t', \ell time(\alpha') + e + d) + 8d \geq \ell time(\alpha') + e$: Immediate.
- $i' \in J(\max(t', \ell time(\alpha') + e + d) + 8d + 2d)$: $i' \in J(\max(t', \ell time(\alpha') + e + d) + 8d)$, as shown above, therefore this follow from Lemma 7.1, where $t = \max(t', \ell time(\alpha') + e + d) + 8d$ and $t' = \max(t', \ell time(\alpha') + e + d) + 10d$.
- i' does not fail in $\beta(\max(t', \ell time(\alpha') + e + d) + 8d + d)$, since i' does not fail in $\beta(\max(t', \ell time(\alpha') + e + d) + 8d + 2d)$ as shown above.

Therefore we can conclude that $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap_{i'}$ is mainstream after $\max(t, \ell time(\alpha') + e + d) + 10d$.

Step 2: We have assumed above that $t' < t - 13d$, that is, $t' + 10d < t - d - 2d$. Also, we have assumed that $\ell time(\alpha') + e + 14d < t$, that is, $\ell time(\alpha') + e + d + 10d < t - d - 2d$. Therefore, $\max(t', \ell time(\alpha') + e + d) + 10d < t - 3d$. We now apply Lemma 7.14, where $t = \max(t', \ell time(\alpha') + e + d) + 10d$, $t' = t - d$, and $cm = \ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap_{i'}$:

- $e + 2d \leq \max(t', \ell time(\alpha') + e + d) + 10d$,
- $\max(t', \ell time(\alpha') + e + d) + 10d \leq t - 3d$,
- $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap_{i'}$ is mainstream after $\max(t, \ell time(\alpha') + e + d) + 10d$.

We therefore conclude that $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap_{i'}$ is mainstream after $t - d$.

Step 3: Notice, then, that by assumption $i \in J(t)$ and i does not fail in $\beta(t - d)$. Therefore by the definition of mainstream, $\ell state(\beta(\max(t', \ell time(\alpha') + e + d) + 8d)).cmap_{i'} \leq \ell state(\beta(t -$

$d)).cmap_i$. Lemma 7.6 then implies that $\ell state(\beta(t-d)).cmap_i \leq \ell state(\alpha'').cmap_i$, since $\beta(t-d)$ is a prefix of α''. Therefore, $\ell state(\beta(\max(t', \ell time(\alpha')+e+d)+8d)).cmap_{i'} \leq \ell state(\alpha'').cmap_i$. Since $\ell state(\beta(\max(t', \ell time(\alpha')+e+d)+8d)).cmap(k_1)_{i'} = \pm$ (as shown above), this means that $\ell state(\alpha'').cmap(k_1)_i = \pm$, as claimed above, concluding Step 3.

This claim that $\ell state(\alpha'').cmap(k_1)_i = \pm$, though, leads to a contradiction: by assumption of this lemma, $\ell state(\alpha'').cmap(k_1)_i = c_1$. Therefore, we conclude that our assumption that $t' < t - 13d$ is incorrect: that is, we must have $t' \geq t - 13d$. That is, we have shown that the upgrade-ready(k_2) event occurs at most $13d$ prior to time t.

We now apply Lemma 7.17, where $c = c_2$, $k = k_2$, and $t = t'$, to conclude that there exists a read-quorum, R, and a write-quorum, W, of configuration c_1 such that no node in $R \cup W$ fails in $\beta(\max(t', \ell time(\alpha')+e)+16d)$. Above we showed that $t' + 13d \geq t$, therefore $t' + 16d \geq t + 3d$, which implies that $\max(t', \ell time(\alpha')+e)+16d \geq t + 3d$. Therefore, we can conclude that there exists a read-quorum, R, and a write-quorum, W, of configuration c_1 such that no node in $R \cup W$ fails in $\beta(t+3d)$. $\square$

The next two lemmas claim that every configuration-upgrade operation completes soon after it begins, or soon after the network stabilizes. The first lemma handles the case where the upgrade begins before the network stabilizes, or during stabilization. The second lemma handles the general case, for all t.

Lemma 7.24 *Let α be an α'-normal execution satisfying: (i)* (α', e)-join-connectivity*, (ii)* (α', e)-recon-readiness*, (iii)* $(\alpha', 2d)$-recon-spacing-1*, (iv)* $(\alpha',\ e,\ 22d)$-configuration-viability.

Assume that $t \in \mathbb{R}^{\geq 0}$ is a time such that $t \leq \ell time(\alpha') + e + 14d$, and that a cfg-upgrade$(k)_i$ *occurs at time t at node i. Assume that node $i \in J(t)$ and that i does not fail in* $\beta(\max(t, \ell time(\alpha')+d)+4d)$.

Then a cfg-upg-ack$(k)_i$ *occurs no later than time* $\max(t, \ell time(\alpha')+d)+4d$.

Proof. Let γ be the configuration-upgrade operation associated with the cfg-upgrade(k) action. Lemma 7.19 shows that proving the following is sufficient to prove the lemma: for every configuration in *removal-set*(γ) there exists a read-quorum, R and a write-quorum, W, such that no node in $R \cup W$ fail by time $\max(t, \ell time(\alpha')+d)+3d$.

Consider any configuration, c_1 with index k_1 in *removal-set*(γ). If t_1 is the time at which configuration $c(k_1+1)$ is installed, configuration-viability ensures that configuration c_1 does not fail until $\max(t_1, \ell time(\alpha')+e)+22d$. Notice that $\ell time(\alpha') + e + 22d > t + 3d$, since $t \leq \ell time(\alpha') + e + 14d$. Therefore, this guarantees that there exists a read-quorum, R, and a write-quorum, W for configuration c_1 such that no node in $R \cup W$ fails until after $\ell time(\alpha') + e + 22d > \max(t, \ell time(\alpha')+d)+3d$. $\square$

Lemma 7.25 *Let α be an α'-normal execution satisfying: (i)*(α', e)-join-connectivity*, (ii)* (α', e)-recon-readiness*, (iii)* $(\alpha', 2d)$-recon-spacing-1*, (iv)* $(\alpha',\ e,\ 22d)$-configuration-viability.

Assume that $t \in \mathbb{R}^{\geq 0}$ is a time, and that a cfg-upgrade$(k)_i$ *occurs in α at time t at node i. Assume that node $i \in J(t)$ and that i does not fail in* $\beta(\max(t, \ell time(\alpha')+e+d)+4d)$.

Then a cfg-upg-ack$(k)_i$ *occurs no later than time* $\max(t, \ell time(\alpha')+e+d)+4d$.

Proof. We prove this lemma by proving a stronger statement by strong induction on the number of cfg-upgrade events in α: if a cfg-upgrade$(*)_j$ event occurs in α at some time $t_{upg} \leq t$ at some node $j \in J(t_{upg})$, and j does not fail in $\beta(\max(t_{upg}, \ell time(\alpha')+e+d)+4d)$, then a matching cfg-upg-ack$(*)_j$ occurs no later than time $\max(t_{upg}, \ell time(\alpha')+e+d)+4d$.

As this is strong induction, we now examine the inductive step. Consider configuration-upgrade γ, the $k+1^{st}$ upgrade operation in α that occurs at time $t_{upg} \leq t$ at node $j \in J(t)$ that does not fail in $\beta(\max(t_{upg}, \ell time(\alpha') + e + d) + 4d)$. Assume, inductively, that if γ' is one of the first k upgrade operations that occurs at time $t' \leq t$ at some node $j' \in J(t')$ that does not fail in $\beta(\max(t', \ell time(\alpha') + e + d) + 4d)$, then a matching cfg-upg-ack$(*)$ occurs no later than time $\max(t', \ell time(\alpha') + e + d) + 4d$. There are two cases to consider.

Case 1: $t_{upg} \leq \ell time(\alpha') + e + 14d$.
Recall that the cfg-upgrade event occurs at node $j \in J(t_{upg})$ where j does not fail in $\beta(\max(t_{upg}, \ell time(\alpha') + e + d) + 4d)$. Lemma 7.24 shows that a cfg-upg-ack$(k)_j$ occurs by time $\max(t_{upg}, \ell time(\alpha') + d) + 4d \leq \max(t_{upg}, \ell time(\alpha') + e + d) + 4d$.

Case 2: $t_{upg} > \ell time(\alpha') + e + 14d$.
Lemma 7.19 shows that proving the following is sufficient to prove the lemma: for every configuration in *removal-set*(γ) there exists a read-quorum, R and a write-quorum, W, such that no node in $R \cup W$ fails in $\beta(\max(t_{upg}, \ell time(\alpha') + d) + 3d)$. Let α'' be the prefix of α ending with the cfg-upgrade event γ. Fix some configuration $c \in$ *removal-set*(γ) with index k; that is, $\ell state(\alpha'').cmap(k)_j = c$. We now apply Lemma 7.23, where $c_1 = c$, $k_1 = k$, α'' is as just defined, and $t = t_{upg}$:

- $t_{upg} > \ell time(\alpha'') + e + 14d$.
- $t_{upg} = \ell time(\alpha'')$.
- $\ell state(\alpha'').cmap(k)_j = c$, since $c \in$ *removal-set*(γ) and α'' is the execution ending with the event γ.
- $j \in J(\max(t_{upg}, \ell time(\alpha') + e + 2d))$, since $j \in J(t_{upg})$ and $t_{upg} > \ell time(\alpha') + e + 14d$.
- Upgrades-Complete Hypothesis: for every cfg-upgrade$(*)_j$ event that occurs in α at some time $t' < t_{upg}$ at some node $j' \in J(\max(t_{upg}, \ell time(\alpha') + e + 2d))$ where j' does not fail in $\beta(\max(t_{upg}, \ell time(\alpha') + e + d) + 4d)$, a matching cfg-upgrade$_{j'}$ occurs by time $\max(t_{upg}, \ell time(\alpha') + e + d) + 4d$: this is the inductive hypothesis, since any cfg-upgrade occuring at time $t' < t_{upg}$ must be one of the first k upgrade events.

Therefore, we conclude that there exists a read-quorum, $R \in$ *read-quorums*(c), and a write-quorum, $W \in$ *write-quorums*(c), such that no node in $R \cup W$ fails in $\beta(t + 3d)$. Since this is true for all $c \in$ *removal-set*(γ), this then shows the desired result.

$\square$

We next present two corollaries that follow from these lemmas. First, we present the unconditional version of Lemma 7.23:

Corollary 7.26 *Let α be an α'-normal execution satisfying (i)* (α', e)-join-connectivity, *(ii)*(α', e)-recon-readiness, *(iii)* $(\alpha', 2d)$-recon-spacing-1, *(iv)* $(\alpha', e, 22d)$-configuration-viability.

Assume that $t \in R^{\geq 0}$ is a time. Assume that c is a configuration, and for some finite prefix α'' of α where $t = \ell time(\alpha'')$, some node $i \in J(t)$ that does not fail in α'', for some index k, $\ell state(\alpha'').cmap(k)_i = c$.

Then there exists a read-quorum, R, and a write-quorum, W, such that no node in $R \cup W$ fails in $\beta(\max(t, \ell time(\alpha') + e + d) + 3d)$.

Proof. If $t > \ell time(\alpha') + e + 14d$, then we show that the result follows from Lemma 7.25 and Lemma 7.23. We apply Lemma 7.25 where $c_1 = c$, $k_1 = k$: notice that Lemma 7.23 assumes that:

- $t > \ell time(\alpha') + e + 14d$: By assumption.
- $t = \ell time(\alpha'')$: By assumption.
- $\ell state(\alpha'').cmap(k)_i = c$: By assumption.
- $i \in J(\max(t, \ell time(\alpha') + e + 2d))$: $t > \ell time(\alpha') + e + 14d$.
- i does not fail in α'': By assumption.
- Upgrade-Completes Hypothesis: Fix some $\mathsf{cfg\text{-}upgrade}(*)_j$ event that occurs at time $t_{upg} < t$ at node $j \in J(\max(t_{upg}, \ell time(\alpha') + e + 2d)$ where j does not fail in $\beta(\max(t_{upg}, \ell time(\alpha') + e + d) + 4d)$. We apply Lemma 7.25, where $t = t_{upg}$ and $i = j$. (Notice that $j \in J(t_{upg})$ by Lemma 7.1.) We therefore conclude that a $\mathsf{cfg\text{-}upgrade}(*)_j$ occurs no later than $\max(t_{upg}, \ell time(\alpha') + e + d) + 4d$, as required by the conclusion of the Upgrade-Completes Hypothesis.

We thus conclude that there exists a read-quorum, $R \in \textit{read-quorums}(c)$ and a write-quorum, $W \in \textit{write-quorums}(c)$ such that no node in $R \cup W$ fails in $\beta(t+3d)$. Since $t > \ell time(\alpha') + e + 14d$, this implies that no node in $R \cup W$ fails in $\beta(\max(t, \ell time(\alpha') + e + d) + 3d)$.

Alternatively, if $t \leq \ell time(\alpha') + e + 14d$, configuration-viability guarantees that there exists a read-quorum, R, and a write-quorum, W, such that no node in $R \cup W$ fails in $\beta(\ell time(\alpha') + e + 22d)$, and again the result follows. □

The second corollary guarantees the liveness of the system; that is, the following corollary shows that read and write operations always terminate eventually:

Corollary 7.27 *Let α be an α'-normal execution satisfying (i)* (α', e)-join-connectivity, *(ii)*(α', e)-recon-readiness, *(iii)* $(\alpha', 2d)$-recon-spacing-1, *(iv)* $(\alpha', e, 22d)$-configuration-viability.

Assume that $t \in R^{\geq 0}$. Assume that at time t, for some $i \in J(t)$ that does not fail in α[7]*, a read_i or write_i occurs in α. Then the operation eventually completes.*

Proof. The read or write operation completes if each phase of the operation completes. Let ψ be the read_i, write_i, $\mathsf{query\text{-}fix}_i$, or recv_i action that sets $op.cmap$ to $cmap$, beginning the phase. Each phase completes when for all $\ell : op.cmap(\ell)_i \in C$, i has sent a gossip message to an appropriate quorum of nodes in $c(\ell)$, and received a response. The only way an operation can fail to terminate, then, is if there does not exist a non-failed read-quorum or a write-quorum of some configuration in $op.cmap$.

Assume that c is a configuration with index k such that $op.cmap(k)_i$ is set to c at some time t' after ψ, and before the phase completes. Then for some α'' where $t' = \ell time(\alpha'')$, $\ell state(\alpha'').cmap(k)_i = c$, since $op.cmap$ is set by copying a truncated version of $cmap_i$. By Corollary 7.26, there exists a read-quorum, R, and a write-quorum, W, such that no node in $R \cup W$ fails in $\beta(\max(t, \ell time(\alpha') + e + d) + 3d)$. No later than time $\max(t, \ell time(\alpha') + e + d) + d$, node i sends a gossip message to every node in $R \cup W$. By time $\max(t, \ell time(\alpha') + e + d) + 2d$ the message is received by every node in $R \cup W$, and each node sends a response to i. By time $\max(t, \ell time(\alpha') + e + d) + 3d$, node i receives the response, and $R \cup W \subseteq acc$. Therefore, for all configurations the read and write quorums survive long enough, and so the phase completes. □

[7]More specifically, we are assuming that i does not fail until after the operation terminates; since we do not here bound how long the operation may take, we instead assume that i does not fail in α. Obviously i failing after the operation completes has no effect on the operation completing.

7.8 Read-Write Latency Results

In this section we state and prove the main result of the latency analysis: if an execution contains a period of time of good behavior, and if this section of the executions is $22d$-configuration-viable, then all read and write operations terminate, and terminate within $8d$. Notice that in the original RAMBO paper, a similar result required the stronger assumption of ∞-*configuration-viability*, an arbitrarily unbounded failure assumption, depending on events earlier in the execution. Here there is no dependency on earlier events: the algorithm is guaranteed to stabilize rapidly, as soon as the network stabilizes.

We need one more lemma. This lemma shows that once a $\mathsf{report}(c)$ action occurs for some configuration with index k, then soon every node has set $cmap(\ell) \neq \bot$, for all $\ell \leq k$. This will allow us to show that if a read or write operation begins long enough after a certain $\mathsf{report}(c)$ operation, then it cannot be interrupted by learning about new configurations with smaller indices.

Lemma 7.28 *Let α be an α'-normal execution satisfying: (i)* (α,e)-join-connectivity*, (ii)* (α', e)-recon-readiness*, (iii)* 6d-recon-spacing*, (iv) (α', e, $4d$)-configuration-viability.*

Assume that α contains decide *events for infinitely many configurations. Let ℓ be a configuration index. Let c_1 be the configuration with index ℓ, and c_2 be the configuration with index $\ell + 1$.*

Let i be the node at which the first $\mathsf{recon}(c_1, c_2)$ *event, π, occurs. Let t be the time at which the* $\mathsf{report}(c_1)_i$ *event, ϕ, occurs.*

Then there exists a CMap, cm, such that:

1. *$cm(\ell) \neq \bot$, and*
2. *cm is mainstream after* $\max(t, \ell time(\alpha') + e + d) + 6d$.

Proof. There are two cases to consider. In each case, we first demonstrate an appropriate cm: we identify a node that performs a $\mathsf{report}(c_1)$ and does not fail too soon. We then show that the cmap of that node is mainstream after $\max(t, \ell time(\alpha') + e + d) + 6d$.

Case 1: $\mathsf{recon}(c_1, c_2)_i$ occurs at some time $\leq \ell time(\alpha') + e + 2d$.

In this case, we use the Recon-Spacing-2 assumption to identify a node with an appropriate cmap, and then use configuration-viability to show that this node survives long enough for its cmap to become mainstream after $\ell time(\alpha') + e + 4d$, which then allows us to show that its cmap is mainstream after $\max(t, \ell time(\alpha') + e + d) + 6d$.

By the Recon-Spacing-2 assumption, there exists a write-quorum, $W \in \mathit{write\text{-}quorums}(c_1)$, such that for every node $j \in W$, a $\mathsf{report}(c_1)_j$ occurs in α prior to π, the recon event that proposes configuration c_2. By configuration-viability, there exists some node $j \in W$ that does not fail by time $\ell time(\alpha') + e + 4d$, since there exists some read-quorum, R, that does not fail by time $\ell time(\alpha') + e + 4d$, and by assumption $R \cap W \neq \emptyset$.

We now show that $cmap_j$ satisfies Property 1 after $\ell time(\alpha') + e + 2d$. Notice that:

$$\ell state(\beta(time(\pi))).cmap(\ell)_j \neq \bot,$$

since the report action notifies j of the configuration c_1 prior to π. By assumption we know that $time(\pi) \leq \ell time(\alpha') + e + 2d$. Therefore we know that $\ell state(\beta(\ell time(\alpha') + e + 2d)).cmap_j \neq \bot$.

Let $cm = \ell state(\beta(\ell time(\alpha') + e + 2d)).cmap_j$. We know, then, that $cm(\ell) \neq \bot$, as desired.

Next we show that cm is mainstream after $\ell time(\alpha') + e + 4d$. We apply Lemma 7.7, where $i = j$, $t = \ell time(\alpha') + e + 2d$:

- $j \in J(\ell time(\alpha') + e + 4d)$: If $\ell = 0$, then $j = i_0$ and we have, by assumption, that i_0 performs a $\textsf{join-ack}_{i_0}$ at time 0, immediately implying the statement by the definition of J.

  Otherwise, we apply Lemma 7.2, where $h = c_1$, $t' = time(\textsf{recon}(c(\ell-1), c_1))$, and $t = \ell time(\alpha') + e + 2d$. Notice that $\ell time(\alpha') + e + 2d \geq time(\textsf{recon}(c(\ell-1), c_1))$ since $\ell time(\alpha') + e + 2d \geq time(\pi)$, and $time(\pi) \geq time(\textsf{recon}(c(\ell-1), c_1))$. We therefore conclude that $members(c_1) \subseteq J(\ell time(\alpha') + e + 2d)$. In particular, this means that $j \in J(\ell time(\alpha') + e + 2d)$. Next we apply Lemma 7.1, where $t = \ell time(\alpha') + e + 2d$ and $t' = \ell time(\alpha') + e + 4d$ to see that $j \in J(\ell time(\alpha') + e + 4d)$.
- $\ell time(\alpha') + e + 2d \geq \ell time(\alpha') + e$: Immediate.
- j does not fail in $\beta(\ell time(\alpha') + e + 3d)$: as shown above j does not fail in $\beta(\ell time(\alpha') + e + 4d)$, by choice of j and configuration-viability.

We then conclude, since $cm = \ell state(\beta(\ell time(\alpha') + e + 2d)).cmap_j$, that cm is mainstream after $\ell time(\alpha') + e + 4d$.

We next apply Lemma 7.14, where $t = \ell time(\alpha') + e + 4d$, $t' = \max(t, \ell time(\alpha') + e + d) + 6d$, and cm is as defined above:

- $e + 2d \leq \ell time(\alpha') + e + 4d$: Immediate.
- $\ell time(\alpha') + e + 4d \leq \max(t, \ell time(\alpha') + e + d) + 6d - 2d$: Immediate.
- cm is mainstream after $\ell time(\alpha') + e + 4d$: As shown above.

Therefore, we conclude that cm is mainstream after $\max(t, \ell time(\alpha') + e + d) + 6d$, as desired.

Case 2: $\textsf{recon}(c_1, c_2)_i$ occurs at some time $> \ell time(\alpha') + e + 2d$.
We first notice that i has been notified of configuration c_1 and then show that the cmap of i is mainstream after $\max(t, \ell time(\alpha') + e + d) + 6d$.

Notice that $\ell state(\beta(t)).cmap(\ell)_i \neq \bot$, since the $\textsf{report}(c_1)_i$ event notifies i of configuration c_1.

We now apply Lemma 7.7, where i is as defined above and $t = \max(t, \ell time(\alpha') + e + d)$, to show that cm is mainstream after $\max(t, \ell time(\alpha') + e + d) + 2d$:

- $\max(t, \ell time(\alpha') + e + d) + 2d \geq \ell time(\alpha') + e$: Immediate.
- $i \in J(\max(t, \ell time(\alpha') + e + d) + 2d)$: We apply Lemma 7.2, where $h = c_1$, t' is the time at which c_1 is proposed, and $t = \max(t, \ell time(\alpha') + e + d) + 2d$. Notice that $\max(t, \ell time(\alpha') + e + d) + 2d$ is no earlier than the time at which c_1 is proposed, since a $\textsf{report}(c_1)$ occurs prior to $\max(t, \ell time(\alpha') + e + d) + 2d$. Also, $\max(t, \ell time(\alpha') + e + d) + 2d \geq \ell time(\alpha') + e + 2d$. Therefore we conclude that $members(c_1) \subseteq J(\max(t, \ell time(\alpha') + e + d) + 2d)$. This implies that $i \in J(\max(t, \ell time(\alpha') + e + d) + 2d)$.
- i does not fail in $\beta(\max(t, \ell time(\alpha') + e + d) + d)$: We know that i does not fail prior to event π, that is, the $\textsf{recon}(c_1, c_2)_i$ event. By Recon-Spacing-1, we know that $time(\pi) \geq t + 6d$. By assumption of this case, we know that $time(\pi) > \ell time(\alpha') + e + 2d$. Therefore i does not fail in $\beta(\max(t, \ell time(\alpha') + e + d) + d)$.

We therefore conclude that cm is mainstream after $\max(t, \ell time(\alpha') + e + d) + 2d$.

We next apply Lemma 7.14, where $t = \max(t, \ell time(\alpha') + e + d) + 2d$, $t' = \max(t, \ell time(\alpha') + e + d) + 6d$, and cm is as defined above:

- $e + 2d \leq \max(t, \ell time(\alpha') + e + d) + 2d$: Immediate.
- $\max(t, \ell time(\alpha') + e + d) + 2d \leq \max(t, \ell time(\alpha') + e + d) + 6d - 2d$: Immediate.
- cm is mainstream after $time(\pi_\ell)$: As shown above.

Therefore, we conclude that cm is mainstream after $\max(t, \ell time(\alpha') + e + d) + 6d$, as desired.

□

We finally prove the main theorem, showing that read and write operations terminate rapidly. This result requires *12d+ε-recon-spacing*, and is similar to Theorem 8.17 from [13]. This earlier theorem states that in a normal, steady-state case, with good environmental behavior, read and write operations terminate within time $8d$. Although the following theorem allows for more complicated behavior, deviating from the assumption of good environmental assumptions, read and write operations still complete rapidly.

Theorem 7.29 *Let α be an α'-normal execution satisfying: (i)* (α,e)-join-connectivity*, (ii)* (α', e)-recon-readiness*, (iii)* 12d+ε-recon-spacing*, (iv)* (α', e, 22d)*-configuration-viability.*

Let $t > \ell time(\alpha') + e + 17d$, and assume a read or write operation starts at time t at some node i. Assume $i \in J(t+8d)$, and does not fail until the read or write operation completes. Also, assume that α contains decide *events for infinitely many configurations. Then node i completes the read or write operation by time $t + 8d$.*

Proof. Let $c_0, c_1, c_2, \ldots$ denote the infinite sequence of successive configurations decided upon in α; by infinite reconfiguration, this sequence exists. For each $k \geq 0$, let π_k be the first $\mathsf{recon}(c_k, c_{k+1})_*$ event in α, let i_k be the location at which this occurs, and let ϕ_k be the corresponding, preceding $\mathsf{report}(c_k)_{i_k}$ event. (The special case of this notation for $k = 0$ is consistent with our usage elsewhere.)

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

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

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

Next, we show that within zero time of ψ occurring, $cmap(\ell)_i \neq \perp$ for all $\ell \leq k$. It is at this point that the proof diverges from that of Lemma 8.17 from [12].

For the purposes of the next two lemmas, fix any $\ell \leq k$. We apply Lemma 7.28, where ℓ is as fixed above, $t = time(\phi_\ell)$, $\phi = \phi_\ell$, $\pi = \pi_\ell$, $c_1 = c_\ell$,and $i = i_\ell$. We therefore conclude that there exists a CMap cm such that:

1. $cm(\ell) \neq \perp$, and
2. cm is mainstream after $\max(time(\phi_\ell), \ell time(\alpha') + e + d) + 6d$.

We next apply Lemma 7.14, where $t = \max(time(\phi_\ell), \ell time(\alpha') + e + d) + 6d$, $t' = time(\psi)$, and cm is as above, to show that cm is mainstream after $time(\psi)$:

- $e + 2d \leq \max(time(\phi_\ell), \ell time(\alpha') + e + d) + 6d$: Immediate.

- $\max(time(\phi_\ell), \ell time(\alpha') + e + d) + 6d \leq time(\psi) - 2d$: By the way in which k is chosen we know that $time(\phi_k) + 8d < time(\psi)$. Also, $time(\phi_\ell) \leq time(\phi_k)$: either $\ell = k$, or ϕ_ℓ precedes π_ℓ which precedes ϕ_k. By assumption we know that $\ell time(\alpha') + e + 8d < t$, and $t \leq time(\psi)$.

- cm is mainstream after $\max(time(\phi_\ell), \ell time(\alpha') + e) + 6d$: As shown above.

Therefore, we conclude that cm is mainstream after $time(\psi)$. We know that $i \in J(t)$, and $t \leq time(\psi)$, so by Lemma 7.1, $i \in J(time(\psi))$. Also, i does not fail until the read or write operation completes, and therefore either the read or write operation completes at $time(\psi)$ (in which case we have proved the desired bound) or i does not fail in $\beta(time(\psi))$. Therefore by definition of a CMap being mainstream, if cm is mainstream after $time(\psi)$, then $cm \leq \ell state(\beta(time(\psi))).cmap_i$.

Having shown this for fixed $\ell \leq k$, we now know that for all $\ell \leq k$ there exists some CMap, cm, such that $cm(\ell) \neq \perp$ and cm is mainstream after $time(\psi)$, this implies that for all $\ell \leq k$, $\ell state(\beta(time(\psi))).cmap(\ell)_i \neq \perp$. Therefore we have shown that within zero time of ψ occurring, $cmap(\ell)_i \neq \perp$ for all $\ell \leq k$.

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

Finally, by Corollary 7.27, the operation eventually terminates, which guarantees that ever configuration in $op.cmap$ remains viable for long enough. □

This shows that assuming (α', e, $22d$)-*configuration-viability* is sufficient to guarantee that read and write operations terminate quickly. As long as the reconfiguration algorithm can guarantee this level of viability, the RAMBO II algorithm will continue to make progress, regardless of any bad behavior the network may experience. Further, while $22d$ may seem a long period of time to ensure viability, it must be remembered that d is typically a small interval: we have been assuming that d is a single message delay in the network. Note that simply deciding on a new configuration to install might take many intervals of d (in [12], it is bounded by $11d$). Also, this $22d$ bound is fairly conservative: by making stronger assumptions as to who begins configuration-upgrade operations, and how gossip messages propagate information about completed configuration-upgrade operations, it is probably possible to improve this bound. In this paper we are primarily interested in the fact that it is a constant time bound.

8 Implementation and Preliminary Evaluation

Musial and Shvartsman [16] have developed a prototype distributed implementation that incorporates both the original RAMBO configuration management algorithm [12] and the new RAMBO II algorithm presented in this paper. The system was developed by manually translating the Input/Output Automata specification to Java code. To mitigate the introduction of errors during translation, the implementers followed a set of precise rules, similar to [2], that guided the derivation of Java code from Input/Output Automata notation. The system is undergoing refinement and tuning, however an initial evaluation of the performance of the two algorithms has been performed in a local-area setting.

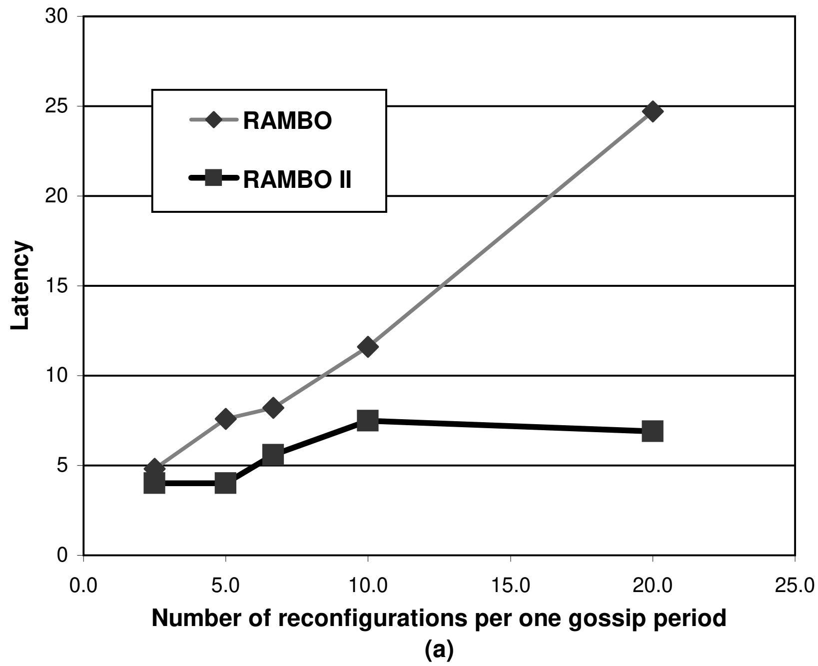

Figure 21: Preliminary empirical evaluation of the average operation latency (measured as the number of gossip intervals), as a function of reconfiguration frequency, measured as number of reconfigurations per one reconfiguration period.

The platform consists of a Beowulf cluster with 13 machines running Linux (Red Hat 7.1). The machines are Pentium processors in the range from 90 MHz to 900 MHz, interconnected via a 100 Mbps Ethernet switch. The implementation of the two algorithms shares most of the code and all low-level routines. Any difference in performance is traceable to the distinct configuration management discipline used by each algorithm.

The machines vary significantly in speed. Given several very slow machines, Musial and Shvartsman do not evaluate absolute performance and instead focus initially on comparing the two algorithms.

The preliminary results in Figure 21 show the average latency of read/write operations as the frequency of reconfigurations grows from about two to twenty reconfigurations per one gossip period. In order to handle such frequent reconfigurations, a large gossip interval (8 seconds) is used. This interval is much larger than the round-trip message delay, thus reducing the effects of network congestion encountered when reconfiguring very frequently. The results show that the overall latency of read/write operations for the new algorithm progressively improve, as the frequency of reconfiguration increases. As expected, the decrease in latency becomes substantial for bursty reconfigurations (at 20 reconfigurations per gossip interval). For less frequent reconfigurations the latency is similar, at about 4 gossip intervals depending on the settings (not shown). This is expected and consistent with our analysis, since the two algorithms are essentially identical when *cmaps* contain one or two configurations. Figure 22 shows the average number of configurations in *cmaps* as a function of reconfiguration frequency. This further explains the difference in performance, since the average number of configurations in *cmaps* is lower in the new algorithm as the frequency of reconfigurations increases.

Finally notice that the modest number of machines used in this study favored the original algo-

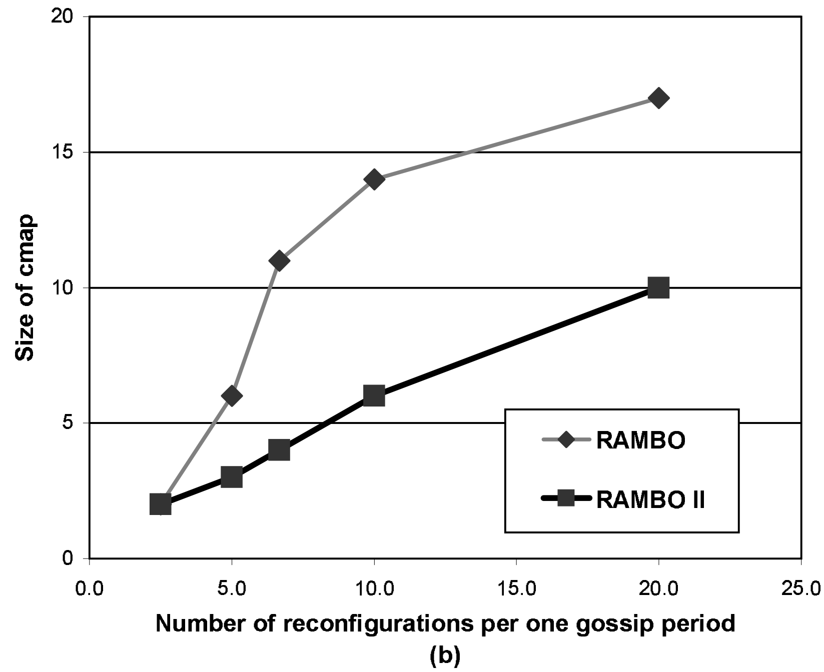

Figure 22: Preliminary empirical evaluation of the average number of configurations in *cmap*'s, as a function of reconfiguration frequency, measured as number of reconfigurations per one reconfiguration period.

rithm. This is because the machines are often members of multiple configurations, thus the number of messages needed to reach fixed-points by the read/write operations of the original algorithm is much lower than is expected when each processor is a member of a few configurations.

Also, notice that this evaluation does not examine the effects of message loss and lack of network connectivity. We hypothesize that, as in the case of frequent bursty reconfiguration, when there are intervals of time in which the network is disconnected, the new algorithm should recover more rapidly. This testing has not yet been performed.

Full performance evaluation is currently in progress. Shvartsman and Musial are investigating how the performance depends on the number of machines and various timing parameters.

9 Conclusion and Open Problems

In this paper we have presented a new algorithm, improving on the original Rambo algorithm by Lynch and Shvartsman [12, 13]. While the original Rambo algorithm is analyzed primarily in the context of good network behavior, we are able to show that our new algorithm functions well even when the network experiences transient periods of bad behavior, including message loss, clock skews, and arbitrary asynchrony, and when reconfiguration is bursty and uneven.

The key to this improvement is a new rapid configuration-upgrade mechanism, which allows the system to stabilize rapidly after a period of bad network behavior. In the previous Rambo algorithm, it might take arbitrarily long to recover from a period of bad behavior. In this new algorithm, however, within a constant time, the system returns to a steady-state condition. This allows the algorithm to function more reliably in a long-running, dynamic system: when a system is expected to function for months and years without failure, it is necessary to rapidly recover from the inevitable transient network failures.

This improvement also makes practical the design of algorithms to choose new configurations. In the earlier version of RAMBO, it is unclear what properties a reconfiguration algorithm must support in order for it to be useful. This paper shows that a reconfiguration automaton must provide exactly $(\alpha', 22d)$-*configuration-viability*.

To design such a reconfiguration algorithm, then, is one of the major open problems posed by this paper. In particular, it seems important to show that if the rate of failure is bounded, then the algorithm continues to make progress. This is similar to the ideas introduced by Karger and Liben-Nowell in [10], in which they assume that the system has a bounded half-life: the time in which either half the processes fail or the number of active processes doubles. Under this assumption, they show that their algorithm operates correctly.

By similarly assuming a bounded rate of failures, it should be possible in certain cases to design a reconfiguration algorithm that guarantees liveness by initiating reconfiguration with some minimum frequency. By choosing appropriate quorums and appropriate numbers of reconfigurations, $(\alpha', 22d)$-*configuration-viability* should be possible.

Other open problems include improving the join protocol, and designing a leave protocol to allow good detection of nodes that have exited the system. Currently, the join protocol is quite simple and it would seem beneficial to require more communication before allowing a node to initiate operations. And when nodes fail or leave, in the algorithm as stated, they are just ignored. By introducing a formal protocol to leave the system, and a method for detecting failed nodes, it might be possible to improve the long-run performance of the system.

Another open problem is to determine how to recover when viability fails (and data is inevitably lost). More generally, is a self-stabilizing version of RAMBO feasible? It would also be interesting to determine whether a version of RAMBO could be adapted to tolerate Byzantine faults.

RAMBO may also allow the construction of other data types, such as weakly consistent memory and sets. It may also be possible to optimize RAMBO to return read values more rapidly, in one phase, in certain cases. An important question would be to determine the most powerful data object that can be implemented using the RAMBO technique; one suspects that it is impossible to implement consensus in this manner.

Finally, it would be interesting to examine how the RAMBO algorithm could be adapted to specific platforms. The algorithm is presented in a fairly abstract fashion. In real implementations, it would be optimized depending on the target platform. In particular, we suspect that RAMBO should work well in sensor networks, mobile-networks, and peer-to-peer networks.

In conclusion, this paper has presented a new algorithm for atomic memory in a highly dynamic environment, proved that is always correct, and presented a set of conditions that guarantee liveness. This provides significant improvements over existing algorithms, rapidly recovering from transient network problems and bursty reconfiguration.

References

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

[2] O. Cheiner and A.A. Shvartsman. Implementing and evaluating an eventually-serializable data service as a distributed system building block. In *Networks in Distributed Computing*,

volume 45 of *DIMACS Series on Disc. Mathematics and Theoretical Computer Science*, pages 43–71. AMS, 1999.

[3] Roberto DePrisco, Nancy Lynch, Alex Shvartsman, Nicole Immorlica, and Toh Ne Win. A formal treatment of Lamport's Paxos algorithm. In progress.

[4] Danny Dolev, Idit Keidar, and Esti Yeger Lotem. Dynamic voting for consistent primary components. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 63–71. ACM Press, 1997.

[5] 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.

[6] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh symposium on Operating systems principles*, pages 150–162, 1979.

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

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

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

[10] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, pages 233–242. ACM Press, 2002.

[11] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufman, 1996.

[12] Nancy Lynch and Alex Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th Intl. Symposium on Distributed Computing*, pages 173–190, 2002.

[13] Nancy Lynch and Alex Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. Technical Report LCS-TR-856, M.I.T., 2002.

[14] Nancy Lynch, Alex Shvartsman, and Roberto De Prisco. Paxos made even simpler (and formal). Manuscript, 2002.

[15] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Twenty-Seventh Annual Intl. Symposium on Fault-Tolerant Computing*, pages 272–281, June 1997.

[16] Peter M. Musial and Alex A. Shvartsman. Implementing a reconfigurable atomic memory service for dynamic networks. submitted for publication.

[17] Roberto De Prisco, Alan Fekete, Nancy A. Lynch, and Alexander A. Shvartsman. A dynamic primary configuration group communication service. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 64–78, September 1999.

[18] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *Transactions on Database Systems*, 4(2):180–209, 1979.

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

[20] P.M.B. Vitányi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 233–243, New York, 1986. IEEE.

Index of Definitions

agreement, 31
atomic, 29

$\beta(t, \alpha)$, 41

$c(k)$, 7, 15
CMap , 15
configuration-viability, 37

d, 36

extend, 14

good execution, 13

in-transit, 15
installed, 35
invariant, 13

$J(t, e, \alpha)$, 36
join-connectivity, 38

mainstream, 41
mainstream after β, 41
mainstream after t, 41

no duplication, 31

operation, 15

$\prec$, 29
prop-cmap(π), 15
prop-phase-start(π), 16

query-cmap(π), 15
query-phase-start(π), 16

$R(\gamma, \ell)$, 16
$R(\pi, k)$, 15
recon-readiness, 39
recon-spacing, 38
removal-set(γ), 16

S, 6, 9
successful recon event, 42

tag(π), 15
truncate, 15
Truncated, 15

update, 14
upgrade-readiness, 39
upgrade-ready, 36
Usable, 15

validity, 31

$W_1(\gamma, \ell)$, 16
$W_2(\gamma)$, 16
$W(\pi, k)$, 15
well-formedness
  consensus, 32
  reader-writer, 14
  recon, 14, 30, 31