Long-Lived Rambo:
Trading Knowledge for Communication

Chryssis Georgiou [*] *Peter M. Musiał* [†] *Alexander A. Shvartsman* [‡]

Abstract

Shareable data services providing consistency guarantees, such as atomicity (linearizability), make building distributed systems easier. However, combining linearizability with efficiency in practical algorithms is difficult. A reconfigurable linearizable data service, called RAMBO, was developed by Lynch and Shvartsman. This service guarantees consistency under dynamic conditions involving asynchrony, message loss, node crashes, and new node arrivals. The specification of the original algorithm is given at an abstract level aimed at concise presentation and formal reasoning about correctness. The algorithm propagates information by means of gossip messages. If the service is in use for a long time, the size and the number of gossip messages may grow without bound. This paper presents a consistent data service for *long-lived* objects that improves on RAMBO in two ways: it includes an incremental communication protocol and a leave service. The new protocol takes advantage of the local knowledge, and carefully manages the size of messages by removing redundant information, while the leave service allows the nodes to leave the system gracefully. The new algorithm is formally proved correct by forward simulation using levels of abstraction. An experimental implementation of the system was developed for networks-of-workstations. The paper also includes selected analytical and preliminary empirical results that illustrate the advantages of the new algorithm.

1 Introduction

This paper presents a practical algorithm implementing long-lived, survivable, atomic read/write objects in dynamic networks, where participants may join, leave, or fail during the course of computation. The survivability of data is ensured through redundancy: the data is replicated and maintained at several network locations. Replication introduces the challenges of maintaining *consistency* among the replicas, and managing *dynamic participation* as the collections of network locations storing the replicas change due to arrivals, departures, and failures of nodes.

A new approach to implementing atomic read/write objects for dynamic networks was developed by Lynch and Shvartsman [1] and extended by Gilbert *et al.* [2]. They developed a memory service called RAMBO (Reconfigurable Atomic Memory for Basic Objects) that maintains atomic, a.k.a linearizable, readable/writable data in highly dynamic environments. In order to achieve availability in the presence of failures, the objects are replicated at several network locations. In order to maintain consistency in the presence of small and transient changes, the algorithm uses *configurations* of locations, 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. Any configuration may be installed at any time. Obsolete configurations can be removed from the system without interfering with the ongoing read and write operations. The algorithm tolerates arbitrary patterns of asynchrony, node crashes, and message loss. It is formally shown [2, 1] that atomicity is maintained

[*]Department of Computer Science, University of Cyprus, CY-1678, Nicosia, Cyprus. Email: chryssis@ucy.ac.cy. The work of this author was performed in part while at the University of Connecticut.

[†]Department of Computer Science and Engineering, University of Connecticut, Storrs CT 06269, USA. Email: piotr@cse.uconn.edu.

[‡]Department of Computer Science and Engineering, University of Connecticut, Storrs CT 06269, USA, and Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA. Email: alex@theory.csail.mit.edu.

in any execution of the algorithm. We developed an experimental implementation of this algorithm on a network-of-workstations [3].

The original RAMBO algorithm is formulated at an abstract level aimed at concise specification and formal reasoning about the algorithm's correctness. Consequently the algorithm incorporates a simple communication protocol that maintain very little protocol state. Nevertheless, showing correctness requires careful arguments involving subtle race conditions [2, 1]. The algorithm propagates information among the participants by means of gossip messages that contain information representing the sender's state. The number and the size of gossip messages may in fact grow without bound. This renders the algorithm impractical for use in *long-lived* applications.

The gossip messages in RAMBO include the set of participants, and the size of these messages increases over time for two reasons. First, RAMBO allows new participants to join the computation, but it does not allow the participants to leave gracefully. In order to leave the participants must pretend to crash. Given that in asynchronous systems failure detection is difficult, it may be impossible to distinguish departed nodes from the nodes that crash. Second, RAMBO gossips information among the participants without regard for what may already be known at the destination. Thus a participant will repeatedly gossip substantial amount of information to others even if it did not learn anything new since the last time it gossiped. While such redundant gossip helps tolerating message loss, it substantially increases the communication burden. Given that the ultimate goal for this algorithm is to be used in long-lived applications, and in dynamic networks with unknown and possibly infinite universe of nodes, the algorithm must be carefully refined to substantially improve its communication efficiency.

Contributions. The paper presents a new algorithm for reconfigurable atomic memory for dynamic networks. The algorithm, called LL-RAMBO, makes implementing atomic survivable objects practical in long-lived systems by managing the knowledge accumulated by the participants and the size of the gossip messages. Each participating node maintains a more complicated protocol state and, with the help of additional local processing, this investment is traded for substantial reductions in the size and the number of gossip messages. Based on [2, 1], we use Input/Output Automata (IOA) [4] to specify the algorithm, then prove it correct in two stages by forward simulation, using levels of abstraction. We include analytical and preliminary empirical results illustrating the advantages of the new algorithm. In more detail, our contributions are as follows.

1. We develop L-RAMBO that implements an atomic memory service and includes a *leave* service (Sect. 3). We prove correctness (safety) of L-RAMBO by forward simulation of RAMBO, hence we show that every trace of L-RAMBO is a trace of RAMBO.

2. We develop LL-RAMBO by refining L-RAMBO to implement *incremental gossip* (Sect. 4). We prove that LL-RAMBO implements the atomic service by forward simulation of L-RAMBO. This shows that every trace of LL-RAMBO is a trace of L-RAMBO, and thus a trace of RAMBO. The proof involves subtle arguments relating the knowledge extracted from the local state to the information that is *not* included in gossip messages. We present the proof in two steps for two reasons: (i) the presentation matches the intuition that the leave service and the incremental gossip are independent, and (ii) the resulting proof is simpler than a direct simulation of RAMBO by LL-RAMBO.

3. We show (Sect. 5) that LL-RAMBO consumes smaller communication resources than RAMBO, while preserving the same read and write operation latency, which under certain steady-state assumptions is at most $8d$ time, where d is the maximum message delay unknown to the algorithm. Under these assumptions, in runs with periodic gossip, LL-RAMBO achieves substantial reductions in communication.

4. We implemented all algorithms on a network-of-workstations. Preliminary empirical results complement the analytical comparison of the two algorithms (Sect. 6).

Background. Several approaches can be used to implement consistent data in (static) distributed systems. Starting with Gifford [5] and Thomas [6], many algorithms used collections of intersecting sets of objects replicas to solve the consistency problem. Upfal and Wigderson [7] use majority sets to emulate shared memory. Vitányi and Awerbuch [8] use matrices of registers where the rows and the columns are written and respectively read by specific processors. Attiya, Bar-Noy and Dolev [9] use majorities to implement shared objects in static message passing systems. Extension with reconfigurable quorums have been explored [10, 11]. These systems have limited ability to support long-lived data when the longevity of processors is limited. Virtual synchrony [12], and group communication services (GCS) in general [13], can be used to implement consistent objects, e.g., by using a global totally ordered broadcast. The universe of nodes in a GCS can evolve, however forming a new view is indicated after a single failure and can take a substantial time, while reads and writes are delayed during view formation. In our algorithm, as in [1, 2], reads and writes can make progress during reconfiguration. In the current approach, arbitrary new configurations can be introduced. This yields a more dynamic system compared to [14, 15, 16, 17, 18] that would require that some new quorums include nodes from the old quorums, thus restricting the choice of the new configuration through the static constraints that need to be satisfied even before the reconfiguration.

The work on reconfigurable atomic memory [10, 1, 2] results in algorithms that are more dynamic because they place fewer restrictions on the choice of new configurations and allow for the universe of processors to evolve arbitrarily. However these approaches are based on abstract communication protocols that are not suited for long-lived systems. In this paper we provide a long-lived solutions by introducing graceful processor departures and incremental gossip. The idea of incrementally propagating information among participating nodes has been previously used in a variety of different settings, e.g., [19, 20, 21, 22, 23, 24, 25]. Incremental gossip is also called anti-entropy [26, 27] or reconciliation [28]; these concepts are used in database replication algorithms, however due to the nature of the application they assume stronger assumptions, e.g., ordering of messages.

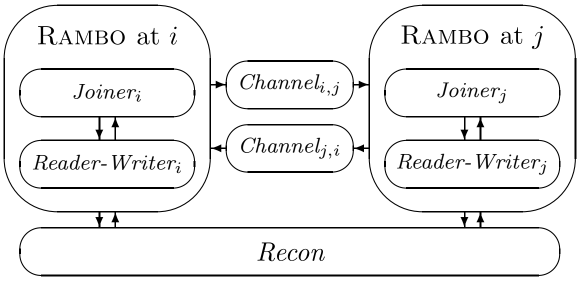

Fig. 1: RAMBO architecture depicting automata at nodes i and j, the channels, and the *Recon* service.

Document structure. In Section 2 we review RAMBO. In Section 3 we specify and prove correct the graceful leave service. Section 4 presents the ultimate system, with leave and incremental gossip, and proves it correct. In Section 5 we present the analytical performance analysis, and in Section 6 we demonstrate the experimental results. We conclude in Section 7. A preliminary version of this paper appears in [29].

2 Reconfigurable Atomic Memory for Basic Objects (RAMBO)

We now describe the RAMBO algorithm as presented in [1], including the rapid configuration upgrade as given in [2]. The algorithm is given for a single object (atomicity is preserved under composition, and multiple objects can be composed to yield a complete shared memory). For the detailed Input/Output Automata code see **Appendix A** or [1, 2]. In order to achieve fault tolerance and availability, RAMBO replicates objects 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*. The quorum intersection property requires that every read-quorum intersect every write-quorum. 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. Any quorum configuration may be installed, and atomicity is preserved in all executions.

The algorithm consists of three kinds of automata: (i) *Joiner* automata, handling join requests, (ii) *Recon* automata, handling reconfiguration requests (recon) and generating a totally ordered sequence of configurations, and (iii) *Reader-Writer* automata, handling read and write requests, manage configuration upgrades, and implement gossip messaging. The overall systems is the composition of these automata with the automata modelling point-to-point communication channels, see Fig. 1.

The *Joiner* automaton is quite straightforward, simply sending a join message when node i joins, and sending a join-ack message whenever a join message is received. The *Recon* automaton establishes a total ordering of configurations; it is implemented using a consensus service (e.g., Paxos [30]), or it can be a much simpler service when the set of possible configurations is finite [31]. Here we assume that a total ordering exists, and we need not discuss this further (for details see [1]).

Domains:
I, a set of processes
V, a set of legal values
C, a set of configurations, each consisting of members, read-quorums, and write-quorums

Input:
$\mathsf{join}(\mathsf{rambo}, J)_i$, J a finite subset of $I - \{i\}$, $i \in I$, such that if $i = i_0$ then $J = \emptyset$
read_i, $i \in I$
$\mathsf{write}(v)_i$, $v \in V$, $i \in I$
$\mathsf{recon}(c, c')_i$, $c, c' \in C$, $i \in members(c)$, $i \in I$
fail_i, $i \in I$

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

Fig. 2: Rambo: External signature.

The external signature of the service is in Fig. 2. A client at node i uses join_i action to join the system. After receiving $\mathsf{join\text{-}ack}_i$, the client can issue read_i and write_i requests, which result in $\mathsf{read\text{-}ack}_i$ and $\mathsf{write\text{-}ack}_i$ responses. The client can issue a recon_i request a reconfiguration. The fail_i action models a crash at node i.

Remark. The fail_i action is an input action coming from the environment. In the original Rambo it is used solely to signal the crash of node i. In this paper we introduce a new kind of action coming from the environment—the leave_i action. This is discussed in detail in the next section. In showing correctness of the new algorithm, we assume that fail_i and leave_i are synonymous: both come from the environment and both result in node i ceasing to participate in the computation. Thus we model the graceful departures as a form of failure. ***Kramer.***

Every node of the system maintains a *tag* and a *value* for the data object. Every time a new value is written, it 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 have two phases, *query* and *propagation*, each accessing certain quorums of replicas. Assume the operation is initiated at node i. First, in the query phase, node i contacts read quorums to determine the most recent known 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 discovered tag 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. Node i then propagates the new tag and the new value to a write quorum. Note that every operation accesses both read and write quorums.

Configurations go through three stages: 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 prior configuration has been notified, the configuration is *installed*. The configuration is *upgraded* when every configuration with a smaller index has been removed. Upgrades are performed by the configuration upgrade operations. Each upgrade operation requires two phases, a query phase and a propagate phase. The first phase contacts a read-quorum and a write-quorum from the old configurations, and the second phase contacts a write-quorum from the new configuration. All three operations, *read*, *write*, and *configuration upgrade*, are implemented using gossip messages.

The *cmap* is a mapping from integer indices to configurations $\cup\{\perp, \pm\}$, initially mapping every integer to $\perp$. It records which configurations are active, which have not yet been created, indicated by $\perp$, and which have already been removed, indicated by $\pm$. The total ordering on configurations determined by *Recon* ensures that all nodes agree on which configuration is stored in each position in *cmap*. We define $c(k)$ to be the configuration associated with index k.

The record *op* is used to store information about the current phase of an ongoing read or write operation, while *upg* is used for information about an ongoing configuration upgrade operation. A node can process read and write operations concurrently with configuration upgrade operations. The *op.cmap* subfield records the configuration map associated with the operation. For read or write operations this consists of the node's *cmap* when a phase begins, augmented by any new configurations discovered during the phase. A phase completes when the initiator has exchanged information with quorums from every valid 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 phase. The *acc* subfield accumulates the ids of the nodes from which quorums have responded during the current phase.

Remark. RAMBO uses monotonically increasing phase numbers to detect fresh responses. The new phase number is computed by incrementing the previous phase number. Here we assume that new phase numbers are computed by adding an arbitrary positive integer to the previous phase number. This is sufficient for the correctness of RAMBO, while this substantially simplifies the proof of correctness of our algorithm. ***Kramer.***

Finally, the nodes communicate via point-to-point channels with the following properties: (1) messages are not corrupted, (2) messages may be lost, (3) messages may be delivered in an arbitrary order, (4) messages may be duplicated, and (5) messages are not spontaneously generated by the channels, which means that for each receive event there is a corresponding send event. We denote by $Channel_{i,j}$ the channel from node i to node j.

3 RAMBO with Graceful Leave

Here we augment RAMBO with a *leave service* allowing the participants to depart gracefully. We prove that the new algorithm, called L-RAMBO, implements atomic memory.

Nodes participating in RAMBO communicate by means of gossip messages containing the latest object value and bookkeeping information that includes the set of known participants. RAMBO allows participants to fail or leave without warning. Since in asynchronous systems it is difficult or impossible to distinguished slow or departed nodes from crashed nodes, RAMBO implements gossip to all known participants, regardless of their status. In highly dynamic systems this leads to (a) the size of gossip messages growing without bounds, and (b) the number of messages sent in each round of gossip increasing as new participants join the computation.

L-RAMBO allows graceful node departures by letting a node that wishes to leave the system to send notification messages to an arbitrary subset of known participants. When another node receives such notification, it marks the sender as departed, and stops gossiping to that node. The remaining nodes propagate the information about the departed nodes to other participants, eventually eliminating gossip to nodes that departed gracefully.

Specification of L-RAMBO**.** We interpret the fail_i event as synonymous with the leave_i event—both are inputs from the environment and both result in node i stopping to participate in all operations. The difference between fail_i and leave_i is strictly internal: leave_i allows a node to leave gracefully. The well-formedness conditions of RAMBO and the specifications of $Joiner_i$ and *Recon* remain unchanged. The introduction of the leave service affects only the specification of the $Reader\text{-}Writer_i$ automata. These changes for L-RAMBO are given in Fig. 3, except for the $\boxed{\text{boxed}}$ segments of code that should be disregarded until the ultimate long-lived algorithm LL-RAMBO is presented in Section 4 (we combine the two specifications to avoid unnecessary repetition and simplify presentation).

The signature of $Reader\text{-}Writer_i$ automaton is extended with actions $\mathsf{recv(leave)}_{j,i}$ and $\mathsf{send(leave)}_{i,j}$ used to communicate the graceful departure status. The state of $Reader\text{-}Writer_i$ is extended with new state variables: $departed_i$, the set of nodes that left the system, as known at node i, $leave\text{-}world_i$, the set of nodes that node i can inform of its own departure, once it decides to leave and sets $leave\text{-}world_i$ to $world - departed$.

The key algorithmic changes involve the actions $\mathsf{recv(m)}_{j,i}$ and $\mathsf{send(m)}_{i,j}$. The original RAMBO algorithm gossips message m includes: W the *world* of the sender, v the object and its tag t, *cm* the *cmap*, *pns* the phase number of the sender, and *pnr* the phase number of the receiver that is known to the sender. The gossip message m in L-RAMBO also includes D, a new parameter, initialized to the *departed* set of the sender.

We now detail the leave protocol. Assume that nodes i and j participate in the service, and node i wishes to depart following the leave_i event, whose effects set the state variable $failed_i$ to *true* in $Joiner_i$, $Recon_i$, and $Reader\text{-}Writer_i$. The leave_i action at $Reader\text{-}Writer_i$ (see Fig. 3) also initializes the set $leave\text{-}world_i$ to the identifiers found in $world_i$, less those found in $departed_i$. Now $Reader\text{-}Writer_i$ is allowed to send one leave notification to any node in $leave\text{-}world_i$. This is done by the $\mathsf{send(leave)}_{i,j}$ action that arbitrarily chooses the destination j from $leave\text{-}world_i$. Note that node i may nondeterministically choose the original fail_i action (see Appendix A or [1]), in which case no notification messages are sent (this is the "non-graceful" departure).

```
Signature:
As in RAMBO, plus new actions:
Input : leave_i, recv(leave)_{j,i}
Output : send(leave)_{i,j}

State:
As in RAMBO, plus new states:
leave-world, a finite subset of I, initially ∅
departed, a finite subset of I, initially ∅
  ig ∈ IGMap, initially ∀k ∈ I,
    ig(k).w-known = ∅, ig(k).w-unack = ∅,
    ig(k).d-known = ∅, ig(k).d-unack = ∅,
    ig(k).p-ack = 0

Transitions at i:
Input recv(⟨W, D, v, t, cm, pns, pnr⟩)_{j,i}
Effect:
  if ¬failed ∧ status ≠ idle then
    status ← active
    world ← world ∪ W
    departed ← departed ∪ D
    [h]hrecv-W(j, i, pnr) ← W
    [h]hrecv-D(j, i, pnr) ← D
    ig(j).w-known ← ig(j).w-known ∪ W
    ig(j).w-unack ← ig(j).w-unack − W
    ig(j).d-known ← ig(j).d-known ∪ D
    ig(j).d-unack ← ig(j).d-unack − D
    if pnr > ig(j).p-ack then
      ig(j).w-known ← ig(j).w-known ∪ ig(j).w-unack
      ig(j).w-unack ← world − ig(j).w-known
      ig(j).d-known ← ig(j).d-known ∪ ig(j).d-unack
      ig(j).d-unack ← departed − ig(j).d-known
      ig(j).p-ack ← pnum1
    if t > tag then (value, tag) ← (v, t)
    cmap ← update(cmap, cm)
    pnum2(j) ← max(pnum2(j), pns)
    if op.phase ∈ {query, prop} ∧ pnr ≥ op.pnum then
      op.cmap ← extend(op.cmap, truncate(cm))
      if op.cmap ∈ Truncated then
        op.acc ← op.acc ∪ {j}
      else
        pnum1 ← pnum1 + 1
        op.acc ← ∅
        op.cmap ← truncate(cmap)
    if upg.phase ∈ {query, prop} ∧ pnr ≥ upg.pnum then
      upg.acc ← upg.acc ∪ {j}

input recv(leave)_{j,i}
Effect:
  if ¬failed ∧ status = active then
    departed ← departed ∪ {j}

Output send(⟨W, D, v, t, cm, pns, pnr⟩)_{i,j}
Precondition:
  ¬failed
  status = active
  j ∈ (world − departed)
  ⟨W, D, v, t, cm, pns, pnr⟩ =
    ⟨world −ig(j).w-known, departed −ig(j).d-known,
      value, tag, cmap, pnum1, pnum2(j) ⟩
Effect:
  pnum1 ← pnum1 + 1
  [h]h-msg ← h-msg∪
      ⟨⟨W, D, v, t, cm, pns, pnr⟩, i, j⟩
  [h]hs-world(i, j, pns) ← world
  [h]hs-departed(i, j, pns) ← departed
  [h]hs-wknow(i, j, pns) ← ig(j).w-known
  [h]hs-dknow(i, j, pns) ← ig(j).d-known
  [h]hs-wunack(i, j, pns) ← ig(j).w-unack
  [h]hs-dunack(i, j, pns) ← ig(j).d-unack
  [h]hs-pack(i, j, pns) ← ig(j).p-ack

input leave_i
Effect:
  if ¬failed then
    failed ← true
    departed ← departed ∪ {i}
    leave-world ← world − departed

output send(leave)_{i,j}
Precondition:
  j ∈ leave-world
Effect:
  leave-world ← leave-world − {j}
```

Fig. 3: Modification of $Reader\text{-}Writer_i$ for L-RAMBO, and for LL-RAMBO (the $\boxed{\text{boxed}}$ code).

When $Reader\text{-}Writer_i$ receives a leave notification from node j, it adds j to its $departed_i$ set. Node i sends gossip messages to all nodes in the set $world_i - departed_i$, which including information about j's departure. When $Reader\text{-}Writer_i$ receives a gossip message that includes the set D, it updates its $departed_i$ set accordingly.

Atomicity of L-RAMBO service. The L-RAMBO system is the composition of all $Reader\text{-}Writer_i$ and $Joiner_i$ automata, the *Recon* service, and $Channel_{i,j}$ automata for all $i, j \in I$. We show atomicity of L-RAMBO by forward simulation that proves that any trace of L-RAMBO is also a trace of RAMBO, and thus L-RAMBO implements atomic objects. The proof uses history variables, annotated with the symbol $[h]$ in Fig. 3.

For each i we define $h\text{-}msg_i$ to be the history variable that keeps track of all messages sent by $Reader\text{-}Writer_i$ automata. Initially, $h\text{-}msg_i = \emptyset$ for all $i \in I$. Whenever a message m is sent by i to some node $j \in I$ via $Channel_{i,j}$, we let $h\text{-}msg_i \leftarrow h\text{-}msg_i \cup \{\langle m, i, j\rangle\}$. We define $h\text{-}MSG$ to be $\bigcup_{i\in I} h\text{-}msg_i$. (The remaining history variables are used in reasoning about LL-RAMBO, in Section 4).

The following lemma states that only good messages are sent.

Lemma 3.1: In any execution of L-RAMBO, if m is a message received by node i in a $\mathsf{recv}(m)_{i,j}$ event, then $\langle m, j, i\rangle \in h\text{-}MSG$, and $m \in \{\langle W, D, v, t, cm, pns, pnr\rangle, \mathsf{leave}, \mathsf{join}\}$, where $\langle W, D, v, t, cm, pns, pnr\rangle \in I \times I \times V \times T \times Cmap \times \mathbb{N} \times \mathbb{N}$.

Proof. By the definition of $Channel_{j,i}$, the messages are not corrupted, and for every receive there exists a preceding send event (messages are not manufactured by the channel). Therefore, m must have been sent by some node $j \in I$ in some earlier event of the execution. Hence $\langle m, j, i\rangle \in h\text{-}msg_j$; by definition, $\langle m, j, i\rangle \in h\text{-}MSG$. Messages are sent only in *Reader-Writer*$_j$ automaton's $\mathsf{send}(\langle W, D, v, t, cm, pns, pnr\rangle)_{j,i}$ or $\mathsf{send}(\mathsf{leave})_{j,i}$ events, or in *Joiner*$_j$ automaton's $\mathsf{send}(\mathsf{join})_{j,i}$ event, and by the code of these events $m = \langle W, D, v, t, cm, pns, pnr\rangle$, $m = \mathsf{leave}$, or $m = \mathsf{join}$. □

Definition 3.2 and Theorem 3.3 are used to show the safety of L-Rambo.

Definition 3.2 ([32]): Let A and S be two automata with the same external signature. A **forward simulation** from A to S is a relation $R \subseteq states(A) \times states(S)$ that satisfies the following two conditions:

1. If t is any initial state of A, then there is an initial state s of S such that $s \in \mathrm{R}(t)$, where $\mathrm{R}(t)$ is an abbreviation for $\{s\colon (t, s) \in \mathrm{R}\}$.
2. If t and $s \in \mathrm{R}(t)$ are reachable states of A and S respectively, and if (t, π, t') is a step of A, then there exists an execution fragment of S from s to some $s' \in \mathrm{R}(t')$, having the same trace as step (t, π, t').

Theorem 3.3 ([32]): If there is a simulation from automaton A to automaton S, then $traces(A) \subseteq traces(S)$.

Next we show that L-Rambo implements Rambo, assuming the environment behavior as (informally) described in Sect. 2. Showing well-formedness is straightforward by inspecting the code. The proof of atomicity is based on a forward simulation from L-Rambo to Rambo.

Theorem 3.4: L-Rambo implements atomic read/write objects.

Proof. We show that L-Rambo algorithm simulates Rambo. Specifically, we show that there exists a simulation relation R from L-Rambo to Rambo that satisfies Definition 3.2. Observe that L-Rambo and Rambo have the same external signatures.

We denote by MSG_{RA} the set of messages in the channel automata of Rambo and by MSG_{LR} the set of messages in the channel automata of L-Rambo.

We define the simulation relation R to map:

(α) a state t of L-Rambo to a state s of Rambo so that every "common" state variable has the same value. For example, for node $i \in I$, $t.world_i = s.world_i$, $t.pnum1_i = s.pnum1_i$, $t.cmap_i = s.cmap_i$, etc.

(β) a message $m = \langle W, D, v, t, cm, pns, pnr\rangle \in Channel_{i,j}.MSG_{\mathrm{LR}}$ to a message $m' = \langle W, v, t, cm, pns, pnr\rangle \in Channel_{i,j}.MSG_{\mathrm{RA}}$ so that: (i) $m.v = m'.v$, (ii) $m.t = m'.t$, (iii) $m.cm = m'.cm$, (iv) $m.pns = m'.pns$, (v) $m.pnr = m'.pnr$, and (vi) $m.W = m'.W$.

Recall that the difference between the two algorithms is in the *Reader-Writer* automata. Therefore, in order to show that L-Rambo simulates Rambo, we focus only on transitions related to the *Reader-Writer* automata. (We remind the reader that the detailed code of Rambo is given in Appendix A.) We now show that R satisfies Definition 3.2:

1. If t is an initial state of L-Rambo then there exists an initial state s of Rambo such that $s \in \mathrm{R}(t)$, since all common state variables have the same initial values. For example, $\forall i \in I$, $t.world_i = \emptyset = s.world_i$, $t.pnum1_i = 0 = s.pnum1_i$, etc.
2. Suppose t and s are reachable states of L-Rambo and Rambo respectively such that $s \in \mathrm{R}(t)$ and that (t, π, t'). We show that there exists a state $s' \in \mathrm{R}(t')$ such that there is an execution fragment of Rambo that has same trace as π.
   (a) If $\pi = \mathsf{send}(m)_{i,j}$, $i, j \in I$, where $m = \langle W, D, v, t, cm, pns, pnr\rangle$ then let s' be such that $(s, \mathsf{send}(m')_{i,j}, s')$, where $m' = \langle W, v, t, cm, pns, pnr\rangle$. Both actions have empty trace, since $\mathsf{send}(m)_{i,j}$ (resp. $\mathsf{send}(m')_{i,j}$) is considered internal with respect to the composition of automata that comprises automaton L-Rambo (resp. Rambo).

   From code of L-Rambo we have that $m = \langle W, D, v, t, cm, pns, pnr\rangle$ is placed in the $Channel_{i,j}.MSG_{\mathrm{LR}}$, where $m.W = t.world_i$, $m.v = t.value_i$, $m.cm = t.cmap_i$, $m.pns = t.pns_i$, and $m.pnr = t.pnum2(j)_i$. From the state correspondence, since $\mathsf{send}(m)_{i,j}$ is enabled, $\mathsf{send}(m')_{i,j}$ is also enabled. Furthermore, $\mathsf{send}(m')_{i,j}$ places $m' = \langle W, v, t, cm, pns, pnr\rangle$ in the $Channel_{i,j}.MSG_{\mathrm{RA}}$ where $m'.W = s.world_i$, $m'.v = s.value_i$, $m'.cm = s.cmap_i$, $m'.pns = s.pns_i$, and $m'.pnr = s.pnum2(j)_i$. From the state correspondence of R for t and s we conclude that the message correspondence of R for t' and s' is preserved. Also, the state correspondence for t' and s' is preserved, since $t'.pnum1_i = t.pnum1_i + 1 = s'.pnum1_i + 1 = s'.pnum1_i$ and all other common variables remain unchanged.

(b) If $\pi = \mathsf{recv}(m)_{j,i}$, $i, j \in I$, where $m = \langle W, D, v, t, cm, pns, pnr\rangle$. By Lemma 3.1, $m \in s.h\text{-}MSG$. Let s' be such that $(s, \mathsf{recv}(m')_{j,i}, s')$, where $m' = \langle W, v, t, cm, pns, pnr\rangle$. Both actions have empty trace, since $\mathsf{recv}(m)_{j,i}$ (resp. $\mathsf{recv}(m')_{j,i}$) is considered internal in the composition of automata that comprises automaton L-Rambo (resp. Rambo).

Now, since $m = \langle W, D, v, t, cm, pns, pnr\rangle$ was in $Channel_{i,j}.MSG_{\mathrm{LR}}$, by the message correspondence of R, message $m' = \langle W, v, t, cm, pns, pnr\rangle$ is in $Channel_{i,j}.MSG_{\mathrm{RA}}$ and m and m' have the mapping defined above. By inspection of the code of the algorithms, the only difference between update performed by L-Rambo and Rambo is that L-Rambo updates the $departed_i$ set remaining common variables undergo identical updates, hence the state correspondence for t' and s' is preserved. Also, the message correspondence is not affected, since no messages are sent.

(c) If $\pi = \mathsf{send}(m)_{i,j}$, $i, j \in I$, where $m = \mathsf{leave}$ then let s' be such that $s = s'$ and Rambo does not perform any action. The action π in L-Rambo has an empty trace since it is considered internal in the composition of automata that comprises automaton L-Rambo. By the code of this action only $leave\text{-}world_i$ variable is updated, remaining common state variables of L-Rambo are unchanged. Since Rambo does not perform a step, its state variables are not modified. Hence the state correspondence for t' and s' is preserved.

(d) If $\pi = \mathsf{recv}(m)_{j,i}$, $i, j \in I$, where $m = \mathsf{leave}$. By Lemma 3.1, $m \in s.h\text{-}MSG$. Let s' be such that $s = s'$ and Rambo does not perform any action. The action π in L-Rambo has an empty trace since it is considered internal in the composition of automata that comprises automaton L-Rambo. By the code of this action only $departed_i$ set is updated, remaining common state variables of L-Rambo are unchanged. Since Rambo does not perform a step, its state variables are not modified. Hence the state correspondence for t' and s' is preserved.

(e) If $\pi = \mathsf{leave}_i$, $i \in I$, then let s' be such that $(s, \mathsf{leave}_i, s')$. By examination of the code the state variable $failed_i$ is set to $true$ by both L-Rambo and Rambo, remaining common state variables are not changed. (Algorithm L-Rambo initializes $leave\text{-}world_i$ with the identifiers found in the set $\{world_i - departed_i - \{i\}\}$.) Since, $s'.failed_i = \mathsf{true} = t'.failed_i$, and all remaining common state variables are unchanged, the state correspondence for t' and s' is preserved.

(f) If π is not one of the above actions, then we choose s' such that (s, π, s'). That is, we simulate the same action. By inspection of the code of the algorithms it follows that any state change after π occurred is identical for both algorithms, hence the state correspondence and message correspondence given by R is preserved.

Therefore, R is a simulation mapping from L-Rambo to Rambo per Definition 3.2. Thus, L-Rambo simulates Rambo. Since Rambo implements atomic objects [1, 2], so does L-Rambo. □

Finally, observe from the preconditions of $\mathsf{send}(\langle W, D, v, t, cm, pns, pnr\rangle)_{j,i}$ that if $i \in departed_j$, then send is disabled, i.e., once a node j learns that another node i left the system, j stops gossiping to i. The correctness of the leave service is as follows: if node i is placed in the departed set of node j, then i indeed departed the service.

Theorem 3.5: For all states s of an execution of algorithm L-Rambo, for any $i, j \in I$, if $i \in s.departed_j$ then $i \in s.departed_i$.

Proof. Proof is done by induction on the length of the execution. The base case holds trivially since all sets are empty in the initial state. Assume that the thesis of the lemma holds up to state s and consider step (s, π, s'). Consider the case where $i \in s.departed_j$, $i, j \in I$. By inspection of the code (see Fig. 3), we see that no identifier is ever removed from *departed*. Hence, we have that $i \in s'.departed_j$ and by inductive hypothesis, $i \in s.departed_i$ and hence $i \in s'.departed_i$ as desired. Let us now consider the more interesting case where $i \notin s.departed_j$ and $i \in s'.departed_j$. That is, the case where j adds i in $departed_j$ during step (s, π, s'). By examination of the code, there are only two actions that would make the above case possible:

(a) $\pi = \mathsf{recv}(m)_{i,j}$, $i, j \in I$, where $m = \mathsf{leave}$. By Lemma 3.1, $m \in s.h\text{-}MSG$. Hence there is a preceding $\mathsf{send}(\mathsf{leave})_{i,j}$ event. By inspection of the code we observe that by the time the preconditions of the output $\mathsf{send}(\mathsf{leave})_{i,j}$ are satisfied, the following is already true: $i \in departed_i$. Hence, we have that $i \in s'.departed_i$ as desired.

(b) $\pi = \mathsf{recv}(m)_{k,j}$, $k, j \in I$ $(k \neq j)$, where $m = \langle W, D, v, t, cm, pnr, pns \rangle$, and $i \in D$. By Lemma 3.1, $m \in s.h\text{-}MSG$. Hence there is a preceding $\mathsf{send}(\langle W, D, v, t, cm, pnr, pns \rangle)_{k,j}$ event, where $i \in D$. By the preconditions of this action we have that $i \in \hat{s}.departed_k$ for some state $\hat{s} < s$. By inductive hypothesis, we have that $i \in \hat{s}.departed_i$ and hence $i \in s'.departed_i$ as desired.

This completes the proof. □

4 Rambo with Graceful Leave and Incremental Gossip

Now we present, and prove correct, our ultimate algorithm, called LL-Rambo (Long-Lived Rambo). The algorithm is obtained by incorporating incremental gossip in L-Rambo, so that the size of gossip messages is controlled by eliminating redundant information. In L-Rambo (resp. Rambo) the gossip messages contain sets corresponding to the sender's *world* and *departed* (resp. *world*) state variables at the time of the sending (Fig. 3). As new nodes join the system and as participants leave the system, the cardinality of these sets grows without bound, rendering Rambo and L-Rambo impractical for implementing long-lived objects. The LL-Rambo algorithm addresses this issue. The challenge here is to ensure that only the certifiably redundant information is eliminated from the messages, while tolerating message loss and reordering.

Specification of LL-Rambo**.** We specify the algorithm by modifying the code of L-Rambo. In Fig. 3 the $\boxed{\text{boxed}}$ segments of code specify these modifications. (The lines annotated with $[h]$ in Fig. 3 deal with history variables that are used only in the proof of correctness.) The new gossip protocol allows node i to gossip the information in the sets $world_i$ and $departed_i$ incrementally to each node $j \in world_i - departed_i$. Following j's acknowledgment[1], node i never again includes this information in the gossip messages sent to j, but will include new information that i has learned since the last acknowledgment by j.

To describe the incremental gossip in more detail we consider an exchange of a gossip messages between nodes i and j, where i is the sender and j is the receiver. The sets *world* and *departed* are managed independently and similarly, and we illustrate incremental gossip using just the set *world*. First we define new data types. Let an *incremental gossip identifier* be the tuple $\langle w\text{-}known, d\text{-}known, w\text{-}unack, d\text{-}unack, p\text{-}ack \rangle$, where *w-known*, *d-known*, *w-unack*, and *d-unack* are finite subsets of I, and *p-ack* is a natural number. Let IG denote the set of all *incremental gossip identifiers*. Finally, let *IGMap* be the set of *incremental gossip maps*, defined as the set of mappings $I \rightarrow IG$. We extend the state of the $Reader\text{-}Writer_i$ automaton with $ig_i \in IGMap$. Node i uses $ig(j)_i$ tuple to keep track of the knowledge it has about the information already in possession of, and currently being propagated to, node j (see Fig.3). Specifically, for each $j \in world_i$, $ig(j)_i.w\text{-}known$ is the set of node identifiers that i is assured is a subset of $world_j$, $ig(j)_i.w\text{-}unack$ is the set of node identifiers, a subset of $world_i$, that j needs to acknowledge. The components $ig(j)_i.d\text{-}known$ and $ig(j)_i.d\text{-}unack$ are defined similarly for the *departed* set. Lastly, $ig(j)_i.p\text{-}ack$ is the phase number of i when the last acknowledgment from j was received. Initially each of these sets is empty, and *p-ack* is zero for each $ig(j)_i$ with $j \in I$.

Node j *acknowledges* a set of identifiers by including this set in the gossip message, or by sending a phase number of i such that node i can deduce that node j received this set of identifiers in some previous message from i to j. Messages that include i's phase number that is larger than $ig(j)_i.p\text{-}ack$ are referred to as *fresh* or *acknowledgment* messages, otherwise they are referred to as *stale* messages. (This is discussed later.)

[1] We note that this is not an explicit acknowledgment of a message, but some future message that contains information about what that node learned.

In Rambo, once node i learns about node j, it can gossip to j at any time. We now examine the $\mathsf{send}(\langle W, D, v, t, cm, pns, pnr\rangle)_{i,j}$ action. The world component, W, is set to the difference of $world_i$ and the information that i knows that j has, $ig(j)_i.w\text{-}known$, at the time of the send. Remaining components of the gossip message are the same as in L-Rambo. The effect of the send action causes phase number of the sender to increase; this ensures that each message sent is labeled with a unique phase number of the sender.

Now we examine $\mathsf{recv}(\langle W, D, v, t, cm, pns, pnr\rangle)_{i,j}$ action at j (note that we switch i and j relative to the code in Fig. 3 to continue referring to the interaction of the sender i and receiver j). The component W contains a subset of node identifiers from j's *world*. Hence W is always used to update $world_j$, $ig(i)_j.w\text{-}known$, and $ig(i)_j.w\text{-}unack$. The update of $world_j$ is identical to that in L-Rambo. By definition $ig(i)_j.w\text{-}known$ is the set of node identifiers that j is assured that i has, hence we update it with information in W. Similarly, by definition $ig(i)_j.w\text{-}unack$ is the set of node identifiers that j is waiting for i to acknowledge. It is possible that i has learned some or all of this information from other nodes and it is now a part of W, hence we remove any identifiers in W that are also in $ig(i)_j.w\text{-}unack$ from $ig(i)_j.w\text{-}unack$; these identifiers do not need further acknowledgment.

What happens next in the effect of recv depends on the value of pnr (the phase number that i believes j to be in). First, if $pnr \leq ig(i)_j.p\text{-}ack$, this means that this message is a stale message since there must have been a prior message from j to i that included phase number of j higher or equal to pnr. Hence, no updates take place. Second, if $pnr > ig(i)_j.p\text{-}ack$, this message is considered to be an acknowledgment message. By definition $ig(i)_j.p\text{-}ack$ contains the phase number of j when last acknowledgment from i was received. Following last acknowledgment, phase number of j was incremented, $ig(i)_j.p\text{-}ack$ was assigned the new value of phase number of j, and lastly new set of identifiers to be propagated was recorded. Since node i replied to j with phase number larger than $ig(i)_j.p\text{-}ack$ it means that j and i exchanged messages where i learned about the new phase number of j, by the same token i also learned the information included in these messages. (We show formally that $ig(i)_j.w\text{-}unack$ is always a subset of each message component W that is sent to i by j.) Hence, it is safe for j to assume that i at least received the information in $ig(i)_j.w\text{-}unack$ and to add it to $ig(i)_j.w\text{-}known$.

Since the choice of i and j is arbitrary, gossip from j to i is defined identically.

Atomicity of LL-Rambo. We show that any trace of LL-Rambo is a trace of L-Rambo, and thus a trace of Rambo. We start by defining the remaining history variables used in the proofs. These variables are annotated in Fig. 3 with a $[h]$ symbol.

- For every tuple $\langle m, i, j\rangle \in h\text{-}msg_i$, where $m = \langle W, D, v, t, cm, pns, pnr\rangle$ and $pns = p$, the history variable $hsent\text{-}W(i, j, p)$ is a mapping from $I \times I \times \mathbb{N}$ to $2^I \cup \{\perp\}$. This variable records the *world* component of the message, W, when i sends message m to j, and i's phase number is p. Similarly, we define a derived history variable $hsent\text{-}D(i, j, p)$, a mapping from $I \times I \times \mathbb{N}$ to $2^I \cup \{\perp\}$. This history variable records the *departed* component of the message, D, when i sends message m to j, and i's phase number is p.

Now we list history variables used to record information for each $\mathsf{send}(\langle W, D, v, t, cm, pns, pnr\rangle)_{i,j}$ event.

- Each of the following variables is a mappings from $I \times I \times \mathbb{N}$ to $2^I \cup \{\perp\}$. $hs\text{-}world(i, j, pns)$ records the value of $world_i$, $hs\text{-}departed(i, j, pns)$ records the value of $departed_i$, $hs\text{-}wknow(i, j, pns)$ records the value of $ig(j)_i.w\text{-}known$, $hs\text{-}dknow(i, j, pns)$ records the value of $ig(j)_i.d\text{-}known$, $hs\text{-}wunack(i, j, pns)$ records the value of $ig(j)_i.w\text{-}unack$, and $hs\text{-}dunack(i, j, pns)$ records the value of $ig(j)_i.d\text{-}unack$.
- $hs\text{-}pack(i, j, pns)$ is a mapping from $I \times I \times \mathbb{N}$ to $\mathbb{N}$. It records the value of $ig(j)_i.p\text{-}ack$.

The last history variables record information in messages at each $\mathsf{recv}(\langle W, D, v, t, cm, pns, pnr\rangle)_{j,i}$ event.

- Each of the following is a mapping from $I \times I \times \mathbb{N}$ to $2^I \cup \{\perp\}$. $hrecv\text{-}W(j, i, pns)$ records the component W (*world*) and $hrecv\text{-}D(j, i, pns)$ records the component D (*departed*).

Similarly as in Section 3, we define history variable $h\text{-}MSG$ that keeps track of messages sent by *Reader-Writer* automata.

Lemma 4.1: In any execution of LL-Rambo, if m is a message received by node i in a $\mathsf{recv}(m)_{i,j}$ event, then $\langle m, j, i\rangle \in h\text{-}MSG$, and $m \in \{\langle W, D, v, t, cm, pns, pnr\rangle, \mathsf{leave}, \mathsf{join}\}$, where $\langle W, D, v, t, cm, pns, pnr\rangle \in I \times I \times V \times T \times Cmap \times \mathbb{N} \times \mathbb{N}$.

Proof. This proof is identical to that of Lemma 3.1, since the format of messages sent by *Reader-Writer* automata in LL-Rambo is as in L-Rambo. □

We continue by showing properties of messages delivered by *Reader-Writer* processes.

Lemma 4.2: Consider a step $\langle s, \pi, s' \rangle$ of an execution α of LL-Rambo, where $\pi = \mathsf{recv}(\langle W, D, v, t, cm, p_j, p_i \rangle)_{j,i}$ for $i, j \in I$, and $p_i > s.ig(j)_i.p\text{-}ack$. Then, (a) $s.ig(j)_i.p\text{-}ack = s.hs\text{-}pack(i, j, p_i)$, (b) $s.ig(j)_i.w\text{-}unack \subseteq s.hs\text{-}wunack(i, j, p_i)$, and (c) $s.ig(j)_i.d\text{-}unack \subseteq s.hs\text{-}dunack(i, j, p_i)$.

Proof. We prove the three parts separately:

Part (a). Assume for contradiction that $p_i > s.ig(j)_i.p\text{-}ack$ and $s.hs\text{-}pack(i, j, p_i) \neq s.ig(j)_i.p\text{-}ack$. By the code of the algorithm and the monotonicity of *pnum1* the only possibility is such that $s.hs\text{-}pack(i, j, p_i) < s.ig(j)_i.p\text{-}ack$. This suggests that there must be a receive event $\mathsf{recv}(\langle W, D, v, t, cm, p_j, p \rangle)_{j,i}$, where $p > s.hs\text{-}pack(i, j, p_i)$ (hence $hrecv\text{-}W(j, i, p)$ and $hrecv\text{-}D(j, i, p)$ are defined) that resulted in the value of $s.ig(j)_i.p\text{-}ack$. By the code of the $\mathsf{recv}_{j,i}$ action, for $i, j \in I$, $ig(j)_i.p\text{-}ack$ is assigned the phase number of i that i has during the receive event. Hence, by the phase number paradigm, we have that $s.ig(j)_i.p\text{-}ack \geq p_i$, which contradicts our initial assumption.

Part (b). From part (a) we have that $hs\text{-}pack(i, j, p_i) = s.ig(j)_i.p\text{-}ack$. From the code it follows that if $ig(j)_i.p\text{-}ack$ does not change then the membership of the set $ig(j)_i.w\text{-}unack$ can only be reduced (the "if $pnr > ig(j)_i.p\text{-}ack$ then" statement in not executed). Therefore, $s.ig(j)_i.w\text{-}unack \subseteq s.hs\text{-}wunack(i, j, p_i)$.

Part (c). Similar to part (b). From part (a) we have that $hs\text{-}pack(i, j, p_i) = s.ig(j)_i.p\text{-}ack$. From the code it follows that if $ig(j)_i.p\text{-}ack$ does not change then the membership of the set $ig(j)_i.d\text{-}unack$ can only be reduced (the "if $pnr > ig(j)_i.p\text{-}ack$ then" statement in not executed). Therefore, $s.ig(j)_i.d\text{-}unack \subseteq s.hs\text{-}dunack(i, j, p_i)$. □

We now state and prove four invariants that lead to the proof of atomicity of LL-Rambo. The first invariant states that a node does not send information to another node that the first node does not already possess.

Invariant 1: For all states s of any execution α of LL-Rambo:
$\langle \langle W, D, v, t, cm, pns, pnr \rangle, i, j \rangle \in s.h\text{-}MSG \Rightarrow W \subseteq s.world_i \wedge D \subseteq s.departed_i$.

Proof. The proof is done by induction on the length of the execution. The base case is trivial because all sets are empty in the initial state. Assume the invariant holds for state s and consider step (s, π, s').

1. $\pi = \mathsf{recv}(\langle W, D, v, t, cm, pns, pnr \rangle)_{j,i}$. By Lemma 4.1, $\langle \langle W, D, v, t, cm, pns, pnr \rangle, j, i \rangle \in h\text{-}MSG$. By inductive hypothesis and the monotonicity of the sets $world_i$ and $departed_i$, in state s', the invariant is maintained.

2. $\pi = \mathsf{recv(join)}_{j,i}$. By Lemma 4.1, $\langle \mathsf{join}, j, i \rangle \in h\text{-}MSG$. By the effects of this action, $s.world_i \subseteq s'.world_i$ and $s.departed_i = s'.departed_i$. By inductive hypothesis and the monotonicity of the sets $world_i$ and $departed_i$, in state s', the invariant is maintained.

3. $\pi = \mathsf{recv(leave)}_{j,i}$. By Lemma 4.1, $\langle \mathsf{leave}, j, i \rangle \in h\text{-}MSG$. By the effects of this action, $s.world_i = s'.world_i$ and $s.departed_i \subseteq s'.departed_i$. By inductive hypothesis and the monotonicity of the sets $world_i$ and $departed_i$, in state s', the invariant is maintained.

4. $\pi = \mathsf{send(m)}_{i,j}$. By the code of this action $m = \langle W, D, v, t, cm, pns, pnr \rangle$, such that $W = s.world_i - s.ig(j)_i.w\text{-}known$ and $D = s.departed_i - s.ig(j)_i.d\text{-}known$. Therefore, $W \subseteq s.world_i$ and $D \subseteq s.departed_i$. Observe that the effects of this action do not change $world_i$ and $departed_i$. Hence $W \subseteq s'.world_i$ and $D \subseteq s'.departed_i$. By inductive hypothesis, the assignment $\langle m, i, j \rangle \in h\text{-}MSG$ maintains the invariant.

5. Other actions do not change the variables involved in the invariant. So, by inductive hypothesis, in state s', the invariant continues to hold.

This completes the proof. □

The following invariant states that the information that i expects j to acknowledge does not exceed the information that actually should be acknowledged.

Invariant 2: For all states s of any execution α of LL-Rambo:
(a) $\forall\ i, j \in I : s.ig(j)_i.w\text{-}unack \subseteq s.world_i - s.ig(j)_i.w\text{-}known$.
(b) $\forall\ i, j \in I : s.ig(j)_i.d\text{-}unack \subseteq s.world_i - s.ig(j)_i.d\text{-}known$

Proof. We prove part (a) of the invariant. The proof of part (b) is analogous: the arguments are made on *departed* related variables instead on *world* related variables.

The proof is done by induction on the length of the execution. The base case is trivial because all sets are empty in the initial state. Assume the invariant (part (a)) holds for state s and consider step (s, π, s').

1. $\pi = \mathsf{recv}(\langle W, D, v, t, cm, pns, pnr\rangle)_{j,i}$. By Lemma 4.1, $\langle\langle W, D, v, t, cm, pns, pnr\rangle, j, i\rangle \in s.h\text{-}MSG$. We consider 4 cases:

   (i) $z \in W \wedge z \notin s.world_i \wedge z \neq j$. This is the first time node i learns about node z, indirectly from j. By initial value of the $ig(z)_i$ record and the monotonicity of $world_i$ the invariant (part (a)) is maintained.

   (ii) $j \notin s.world_i$. This is the first time i learns about j, directly from j. Similarly to case 1(i), $ig(j)_i.w\text{-}unack$ and $ig(j)_i.w\text{-}known$ are initialized to empty. Also $ig(j)_i.p\text{-}ack$ is set to zero. Before the first inner "if-statement", $s'.world_i = s.world_i \cup W$ (note that j is now in $s'.world_i$). Also, $s'.ig(j)_i.w\text{-}known = ig(j)_i.w\text{-}known \cup W$ and $s'.ig(j)_i.w\text{-}unack = ig(j)_i.w\text{-}unack - W$. Since $j \notin s.world_i$, by the code of the send action i never send messages to j, and hence pnr is zero. Therefore, the "if $pnr > ig(j)_i.p\text{-}ack$ then" statement is not executed. By inductive hypothesis (part (a)), the invariant (part (a)) is reestablished.

   (iii) $j \in world_i \wedge pnr \leq s.ig(j)_i.p\text{-}ack$. Since $pnr \leq s.ig(j)_i.p\text{-}ack$, this implies that node i has learned and communicated with node j in some earlier step of the execution. By the effects of this action, $s'.world_i = s.world_i \cup W$, $s'.ig(j)_i.w\text{-}known = s.ig(j)_i.w\text{-}known \cup W$, and $s'.ig(j)_i.w\text{-}unack = s.ig(j)_i.w\text{-}unack - W$. Then,

   $$\begin{aligned} s'.world_i - s'.ig(j)_i.w\text{-}known &= (s.world_i \cup W) - (s.ig(j)_i.w\text{-}known \cup W) \\ &= s.world_i - s.ig(j)_i.w\text{-}known - W. \end{aligned}$$

   By inductive hypothesis (part (a)),

   $$s.ig(j)_i.w\text{-}unack - W \subseteq s.world_i - s.ig(j)_i.w\text{-}known - W.$$

   Therefore,

   $$s'.ig(j)_i.w\text{-}unack \subseteq s'.world_i - s'.ig(j)_i.w\text{-}known.$$

   Thus, the invariant (part (a)), in state s', is maintained.

   (iv) $j \in world_i \wedge pnr > s.ig(j)_i.p\text{-}ack$. Using similar arguments as in case 1(iii) we have that up to the "if $pnr > ig(j)_i.p\text{-}ack$ then" statement the invariant (part (a)) holds. Since $pnr > s.ig(j)_i.p\text{-}ack$ the "if-statement" is executed, and by its effects we have $s'.ig(j)_i.w\text{-}unack = s'.world_i - s'.ig(j)_i.w\text{-}known$. Hence, the invariant (part (a)) is reestablished.

2. $\pi = \mathsf{recv}(\mathsf{join})_{j,i}$. By Lemma 4.1, $\langle \mathsf{join}, j, i\rangle \in s.h\text{-}MSG$. By the code of this action $s.world_i \subseteq s'.world_i$. If this is the first time node i learns about node j then by the initial assignment of the $ig(j)_i$ record and the monotonicity of $world_i$, in state s', the invariant (part (a)) holds. On the other hand, if $j \in s.world_i$ then by the inductive hypothesis (part (a)) and the monotonicity of $world_i$, in state s', the invariant (part (a)) holds.

3. Other actions do not change the variables involved in the invariant (part (a)). So, by inductive hypothesis (part (a)) , in state s', the invariant (part (a)) continues to hold.

This completes the proof of part (a). As mentioned at the beginning of the proof, part (b) is shown in a similar manner. ☐

In the next invariant we use history variables to show that if i sends a message to j when it believes that j's phase number is p then the information that i needs j to acknowledge does not exceed the information included in that message.

Invariant 3: For all states s of any execution α of LL-Rambo:
(a) $\langle\langle W, D, v, t, cm, p, pnr\rangle, i, j\rangle \in s.h\text{-}MSG \Rightarrow s.hs\text{-}wunack(i, j, p) \subseteq W$
(b) $\langle\langle W, D, v, t, cm, p, pnr\rangle, i, j\rangle \in s.h\text{-}MSG \Rightarrow s.hs\text{-}dunack(i, j, p) \subseteq D$

Proof. The proof is done by induction on the length of the execution. The base case is trivial because all sets are empty in the initial state. Assume the invariant holds for state s and consider step (s, π, s').

1. $\pi = \mathsf{send}(\langle W, D, v, t, cm, pns, pnr\rangle)_{i,j}$. By the effects of this action we have that $W = s'.world_i - s'.ig(j)_i.w\text{-}known$ and $s'.hs\text{-}wunack(i, j, p) = s'.ig(j)_i.w\text{-}unack$ is defined. Also, $D = s'.departed_i - s'.ig(j)_i.d\text{-}known$ and $s'.hs\text{-}dunack(i, j, p) = s'.ig(j)_i.d\text{-}unack$ is defined. By Invariant 2 we have that $s'.ig(j)_i.w\text{-}unack \subseteq s'.world_i - s'.ig(j)_i.w\text{-}known$, and $s'.ig(j)_i.d\text{-}unack \subseteq s'.departed_i - s'.ig(j)_i.d\text{-}known$. Thus, by substitution we get that $s'.hs\text{-}wunack(i, j, p) \subseteq W$ and $s'.hs\text{-}dunack(i, j, p) \subseteq D$, as desired. Therefore, by inductive hypothesis, in state s', the invariant is reestablished.

2. Other actions do not change the variables involved in the invariant. So, by inductive hypothesis, in state s', the invariant continues to hold.

This completes the proof. ☐

We use Lemmas 4.1 and 4.2, and Invariants 1, 2, and 3 to show the key Invariant 4 for the atomicity of LL-Rambo. Here we show that node i never overestimates the knowledge possessed by j.

Invariant 4: For all states s of any execution α of LL-Rambo:
(a) $\forall\ i, j \in I : s.ig(j)_i.w\text{-}known \subseteq s.world_j$,
(b) $\forall\ i, j \in I : s.ig(j)_i.d\text{-}known \subseteq s.departed_j$.

Proof. We prove part (a) of the invariant. The proof of part (b) is analogous: the arguments are made on *departed* related variables instead on *world* related variables.

The proof is done by induction on the length of the execution. The base case is trivial because all sets are empty in the initial state. Assume the invariant (part (a)) holds for state s and consider step (s, π, s') (following discussion involves only part(a) of invariant).

1. $\pi = \mathsf{recv}(\langle W, D, v, t, cm, pns, pnr\rangle)_{*,j}$. By Lemma 4.1, $\langle\langle W, D, v, t, cm, pns, pnr\rangle, *, j\rangle \in s.h\text{-}MSG$. The set $s.ig(j)_i.w\text{-}known$, for $i \in I$, $i \neq j$, is not updated by the effects of this action. Observe that $s.world_j \subseteq s'.world_j$ (for $i = j$ the invariant is trivially maintained). By inductive hypothesis and the monotonicity of $world_j$, in state s', the invariant is maintained.

2. $\pi = \mathsf{recv}(\mathsf{join})_{*,j}$. By Lemma 4.1 $\langle \mathsf{join}, *, j\rangle \in s.h\text{-}MSG$. The discussion in this case is identical to that of case 1.

3. $\pi = \mathsf{recv}(\langle W, D, v, t, cm, pns, pnr\rangle)_{j,i}$. By Lemma 4.1, $\langle\langle W, D, v, t, cm, pns, pnr\rangle, j, i\rangle \in s.h\text{-}MSG$. We consider 4 cases:

   (i) $z \in W \wedge z \notin s.world_i \wedge z \neq j$. This is the first time node i learns about node z, indirectly from j. By initial value of the $ig(z)_i$ record and the monotonicity of $world_i$ the invariant is maintained. (This is for all nodes other than j. We consider j in the remaining three subcases.)

   (ii) $j \notin s.world_i$. This is the first time i learns about j, directly from j. Similarly to case 3(i), $ig(j)_i.w\text{-}unack$ and $ig(j)_i.w\text{-}known$ are initialized to empty. Also $ig(j)_i.p\text{-}ack$ is set to zero. Before the "if-statement", $s'.world_i = s.world_i \cup W$ (note that j is now in $s'.world_i$). Also, $s'.ig(j)_i.w\text{-}known = ig(j)_i.w\text{-}known \cup W$ $(ig(j)_i.w\text{-}known = W)$ and $s'.ig(j)_i.w\text{-}unack =$

$ig(j)_i.w\text{-}unack - W$ ($ig(j)_i.w\text{-}unack$ remains empty). Since $j \notin s.world_i$, by the code of the send action i never sent messages to j, and hence pnr is zero. Therefore, the "if $pnr > ig(j)_i.p\text{-}ack$ then" statement is not executed. By substitution $s'.ig(j)_i.w\text{-}known = W$. From Invariant 1 we have that $W \subseteq s.world_j = s'.world_j$. Hence $s'.ig(j)_i.w\text{-}known \subseteq s'.world_j$, as desired. By inductive hypothesis, in s', the invariant is maintained.

(iii) $j \in s.world_i \wedge pnr \leq s.ig(j)_i.p\text{-}ack$. Since $pnr \leq s.ig(j)_i.p\text{-}ack$, this implies that node i has learned and communicated with node j in some earlier step of the execution. By the effects of this action $s'.ig(j)_i.w\text{-}known = s.ig(j)_i.w\text{-}known \cup W$. From Invariant 1 we have that $W \subseteq s.world_j = s'.world_j$. Also by inductive hypothesis $s.ig(j)_i.w\text{-}known \subseteq s'.world_j$. Therefore, $s'.ig(j)_i.w\text{-}known \subseteq s'.world_j$, as desired. Thus, in s', the invariant is maintained.

(iv) $j \in s.world_i \wedge pnr > s.ig(j)_i.p\text{-}ack$. Using similar arguments as in case 3(iii) we have that up to the "if $pnr > ig(j)_i.p\text{-}ack$ then" $ig(j)_i.w\text{-}known = s.ig(j)_i.w\text{-}known \cup W$ and $ig(j)_i.w\text{-}unack = s.ig(j)_i.w\text{-}unack - W$. Since $pnr > s.ig(j)_i.p\text{-}ack$ the "if-statement" is executed; by its effects we get $s'.ig(j)_i.w\text{-}known = ig(j)_i.w\text{-}known \cup ig(j)_i.w\text{-}unack$. Recall that $ig(j)_i.w\text{-}known = s.ig(j)_i.w\text{-}known \cup W$. By Invariant 1 and inductive hypothesis, $ig(j)_i.w\text{-}known \subseteq s'.world_j$.

Since pnr is contained in the message from j by i, we have that $s.hrecv\text{-}W(i,j,pnr)$ must be defined; moreover, by the code of the algorithm, the monotonicity of the *world* variable, and the definition of $hrecv\text{-}W(i,j,pnr)$, we have that $s.hrecv\text{-}W(i,j,pnr) \subseteq s'.world_j$. By the properties of the *Channel* automata we have that $s.hsent\text{-}W(i,j,pnr)$ must also be defined and its value equals the value of $s.hrecv\text{-}W(i,j,pnr)$. By Invariant 3 we have that $s.hs\text{-}wunack(i,j,pnr) \subseteq s.hsent\text{-}W(i,j,pnr)$. By Lemma 4.2(b) we have that if $pnr > s.ig(j)_i.p\text{-}ack$ then $s.ig(j)_i.w\text{-}unack \subseteq s.hs\text{-}wunack(i,j,pnr)$. By substitution we have $s.ig(j)_i.w\text{-}unack \subseteq s'.world_j$.

Therefore, $s'.ig(j)_i.w\text{-}known \subseteq s'.world_j$, as desired. Thus, by inductive hypothesis, in s', the invariant is reestablished.

4. Other actions do not change the variables involved in the invariant. So, by inductive hypothesis, in state s', the invariant continues to hold.

This completes the proof of part (a). As mentioned at the beginning of the proof, part (b) is shown in a similar manner. □

Finally we show the atomicity of objects implemented by LL-RAMBO by proving that it simulates L-RAMBO, i.e., by showing that every trace of LL-RAMBO is a trace of L-RAMBO (hence of RAMBO).

Theorem 4.3: LL-RAMBO implements atomic read/write objects.

Proof. We show that LL-RAMBO simulates L-RAMBO. Specifically, we show that there exists a simulation relation R from LL-RAMBO automaton to L-RAMBO automaton that satisfies Definition 3.2. Observe that LL-RAMBO and L-RAMBO have the same external signatures.

We denote by MSG_{LR} the set of messages in the channel automata of L-RAMBO and by MSG_{LLR} the set of messages in the channel automata of LL-RAMBO.

We define the simulation relation R to map:

(a) a state t of LL-RAMBO to a state s of L-RAMBO so that every "common" state variable has the same value. For example, for node $i \in I$, $t.world_i = s.world_i$, $t.departed_i = s.departed_i$, $t.pnum1_i = s.pnum1_i$, $t.cmap_i = s.cmap_i$, etc.

(b) a message $m = \langle W, D, v, t, cm, pns, pnr\rangle \in Channel_{i,j}.MSG_{\text{LLR}}$ to a message $m' = \langle W, D, v, t, cm, pns, pnr\rangle \in Channel_{i,j}.MSG_{\text{LR}}$ so that: (i) $m.v = m'.v$, (ii) $m.t = m'.t$, (iii) $m.cm = m'.cm$, (iv) $m.pns = m'.pns$, and $m'.W = hs\text{-}world(i,j,pns)$, and lastly (vii) $m.D = hs\text{-}departed(i,j,pns) - hs\text{-}dknow(i,j,pns)$ and $m'.D = hs\text{-}departed(i,j,pns)$. (We assume that the history variables $hs\text{-}world(i,j,pns)$ and $hs\text{-}departed(i,j,pns)$ are used in L-RAMBO in the similar manner that they are used in LL-RAMBO.)

Recall that the difference between the two algorithms is in the *Reader-Writer* automata. Therefore, in order to show that LL-Rambo simulates L-Rambo, we focus only on transitions related to the *Reader-Writer* automata. We now show that R satisfies Definition 3.2:

1. If t is an initial state of LL-Rambo then there exists an initial state s of L-Rambo such that $s \in \mathrm{R}(t)$, since all common state variables have the same initial values. For example, $\forall i \in I$, $t.world_i = s.world_i = \emptyset$, $t.departed_i = s.departed_i = \emptyset$, $t.pnum1_i = s.pnum1_i = 0$, etc. Also, the channels do not contain any messages.

2. Suppose t and s are reachable states of LL-Rambo and L-Rambo respectively such that $s \in \mathrm{R}(t)$ and that (t, π, t'). We show that there exists a state $s' \in \mathrm{R}(t')$ such that there is an execution fragment of L-Rambo that has the same trace as π.

   (a) If $\pi = \mathsf{recv}(m)_{j,i}$, $i, j \in I$, where $m = \langle W, D, v, t, cm, p, pnr\rangle$. By Lemma 4.1, $m \in s.h\text{-}MSG$. Let s' be such that $(s, \mathsf{recv}(m')_{j,i}, s')$. Both actions have empty trace, since $\mathsf{recv}(m)_{j,i}$ (resp. $\mathsf{recv}(m')_{j,i}$) is considered internal with respect to the composition of automata that comprises automaton LL-Rambo (resp. L-Rambo).

   Now, since $m = \langle W, D, v, t, cm, p, pnr\rangle$ was in $Channel_{j,i}.MSG_{\text{LLR}}$, by the message correspondence of R, message $m' = \langle W, D, v, t, cm, p, pnr\rangle$ is in $Channel_{j,i}.MSG_{\text{LR}}$ and m and m' have the mapping defined above. By inspection of the code of both algorithms, it follows that besides state variables $world_i$ and $departed_i$, all other common state variables are updated identically as the information obtained by messages m and m' is the same for these variables. Also, the message correspondence is not affected, since no messages are sent.

   Hence we focus on showing that $t'.world_i = s'.world_i$ and $t'.departed_i = s'.departed_i$. From the code of LL-Rambo, we have that $t'.world_i = t.world_i \cup m.W$ and $t'.departed_i = t.departed_i \cup m.D$. Since m was received (and hence removed from the channel), by the properties of the channel, there was a send event that placed m in the channel so that $m.W = t.hsent\text{-}W(j,i,p) = t.hs\text{-}world(j,i,p) - t.hs\text{-}wknow(j,i,p)$ and $m.D = t.hsent\text{-}D(j,i,p) = t.hs\text{-}departed(j,i,p) - t.hs\text{-}dknow(j,i,p)$. From Invariant 4 and the monotonicity of variables *world* and *departed* we have that $hs\text{-}wknow(j,i,p) \subseteq t.world_i$ and $hs\text{-}dknow(j,i,p) \subseteq t.departed_i$. From this and the fact that $t.hs\text{-}wknow(j,i,p) \subseteq t.hs\text{-}world(j,i,p)$ and $t.hs\text{-}dknow(j,i,p) \subseteq t.hs\text{-}departed(j,i,p)$, we conclude that $t'.world_i = t.world \cup t.hs\text{-}world(j,i,p)$ and $t'.departed_i = t.departed \cup t.hs\text{-}departed(j,i,p)$. But by the state and message correspondence of R we have that $t.world = s.world$, $t.departed = s.departed$, $m'.W = t.hs\text{-}world(j,i,p)$, and $m'.D = t.hs\text{-}departed(j,i,p)$. Hence, $t'.world_i = s.world \cup m'.W = s'.world$ and $t'.departed_i = s.departed \cup m'.D = s'.departed$, as desired.

   (b) If $\pi = \mathsf{send}(m)_{i,j}$, $i, j \in I$, where $m = \langle W, D, v, t, cm, p, pnr\rangle$. Let s' be such that $(s, \mathsf{send}(m')_{i,j}, s')$. Both actions have empty trace, since $\mathsf{send}(m)_{i,j}$ (resp. $\mathsf{send}(m')_{i,j}$) is considered internal in the composition of automata that comprises automaton LL-Rambo (resp. L-Rambo).

   From the code of LL-Rambo we have that $m = \langle W, D, v, t, cm, pns, pnr\rangle$ is placed in $Channel_{i,j}.MSG_{\text{LLR}}$ where $m.W = hs\text{-}world(i,j,pns) - hs\text{-}wknow(i,j,pns)$, $m.D = hs\text{-}departed(i,j,pns) - hs\text{-}dknow(i,j,pns)$, $m.v = t.value_i$, $m.cm = t.cmap_i$, $m.pns = t.pnum1_i$, and $m.pnr = t.pnum2(j)_i$. From the state correspondence, since $\mathsf{send}(m)_{i,j}$ is enabled, $\mathsf{send}(m')_{i,j}$ is also enabled. Furthermore, $\mathsf{send}(m')_{i,j}$ places $m' = \langle W, D, v, t, cm, pns, pnr\rangle$ in $Channel_{i,j}.MSG_{\text{LR}}$ where $m'.W = hs\text{-}world(i,j,pns)$, $m'.D = hs\text{-}departed(i,j,pns)$, $m'.v = s.value_i$, $m'.cm = s.cmap_i$, $m'.pns = s.pnum1_i$, and $m'.pnr = s.pnum2(j)_i$. From the state correspondence of R for t and s we conclude that the message correspondence of R for t' and s' is preserved. Also, the state correspondence for t' and s' is preserved, since $t'.pnum1_i = t.pnum1_i + 1 = s.pnum1_i + 1 = s'.pnum1_i$ and all other common variables remain unchanged.

   (c) If π is an action besides $\mathsf{send}(m)_{i,j}$ or $\mathsf{recv}(m)_{j,i}$, then we choose s' such that (s, π, s'). That is, we simulate the same action. By inspection of the code of the algorithms it follows that any state change after π occurred is identical for both algorithms, hence the state correspondence and message correspondence given by R is preserved.

Therefore, R is a simulation mapping from LL-Rambo to L-Rambo per Definition 3.2. By Theorem 3.3, any trace of LL-Rambo is a trace of L-Rambo. Since L-Rambo implements atomic read/write objects per Theorem 3.4, so does LL-Rambo. □

5 LL-Rambo Performance

In this section we analyze the performance of LL-Rambo. We perform conditional analysis of read and write operation latency — we assess improvement in communication for specific scenarios.

5.1 Conditional read/write operation latency analysis.

A conditional analysis of Rambo read and write operation latency is presented in [2]. Here we show that under the same conditions LL-Rambo has the same operation latency. We start by giving relevant definitions (based on [2] and [32]). Let d denote the maximum message delivery latency. Let d also be the interval at which the gossip messages are sent. An execution with times associated with all events is called a *timed execution*. A timed execution is said to be *admissible* if the following condition holds: "If timed execution ξ is an infinite sequence, then the times of the actions approach ∞. If ξ is a finite sequence, then in the final state of ξ, every enabled task must be allowed to complete" (for more details see [32]).

Let α be an admissible timed execution, and let α' be a finite prefix of α. Let $\ell time(\alpha')$ denote the time of the last event in α'. We say α is an α'-*normal* execution if (i) after α', the local clocks of all automata progress at exactly the rate of real time, (ii) no message sent in α after α' is lost, and (iii) if a message is sent at time t in α and it is delivered, then it is delivered by the time $\max\{t+d, \ell time(\alpha')+d\}$.

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

As with all quorum-based algorithms, operation liveness depends on all the nodes in some quorums remaining alive or not departing. We say that a configuration is *installed* when every member of the configuration has been notified about the configuration. We say that an execution α is (α',e,τ)-*configuration-viable* if for every installed configuration, there exists a read-quorum, R, and a write-quorum, W, such that no process in $R \cup W$ fails or departs before the maximum of (i) time τ after the next configuration is installed, and (ii) $\ell time(\alpha')+e+\tau$.

We say that execution α satisfies (α',τ)-*recon-spacing* if after α', at least time τ elapses between the event that reports a new configuration c ($\mathsf{report}(c)_i$) and any following event that proposes a new configuration ($\mathsf{recon}(c,*)_i$). In other words, after α', when the system stabilizes, reconfigurations are not too frequent.

Execution α is said to satisfy (α',e)-*join-connectivity* if after α', for any two nodes that both joined the system at time $t-e$, they know about each other by time t.

Execution α satisfies $(\alpha',e+\tau)$-*recon-readiness* if after α', every $\mathsf{recon}(c)$ event proposing a new configuration includes a node i in c only if i joined at least time $e+\tau$ ago. This, in conjunction with (α',e)-*join-connectivity*, ensure that all the nodes in active configurations are aware of each other.

As in [2], we assume that α is an α'-*normal* execution, satisfying $(\alpha',e,23d)$-*configuration-viability*, $(\alpha',8d)$-*recon-spacing*, (α',e)-*join-connectivity*, and $(\alpha',e+d)$-*recon-readiness*. (See [2] for a further discussion of these assumptions.) With this we show conditionally that read and write operations take no more than $8d$ time.

Theorem 5.1: Let α be an α'-*normal* execution of LL-Rambo satisfying join-connectivity, recon-readiness, recon-spacing, and configuration-viability. Let $t > \ell time(\alpha')+e+d$. Assume i is a node that received a $\mathsf{join\text{-}ack}_i$ prior to time $t-e-d$, and neither fails nor departs in α until after time $t+8d$. Then if a read or write operations starts at node i time t, it completes by time $t+8d$.

The proof is essentially identical to the proof of Theorem 5.3 in [2]. The key observation is that under the assumed conditions the incremental gossip does not affect the pattern of messages. The only difference is that these messages do not contain information that had been already propagated.

5.2 Communication Efficiency Analysis

In this section we illustrate the communication savings attained by LL-Rambo as compared to Rambo. The savings are assessed both in terms of gossip message size and number of gossip messages. We present communication analysis of α'-*normal* timed executions α, where the prefix α' satisfies specific properties. Following the prefix α', we assume that α is divided into *rounds* of length d, where d is greater than the maximum message latency. The active nodes send gossip messages at the beginning of each round and subsequently receive gossip messages sent during the current round. We assume that in all executions node failure and departures do not disable the quorum systems used by the algorithms.

We observe that given any timed execution α of LL-Rambo, it is possible to construct an execution $\hat{\alpha}$ of Rambo that has the same interaction with its environment as α (as follows from Theorem 4.3), but that includes additional gossip messages and gossip messages with different content, and that excludes leave notification messages that are specific to LL-Rambo. We also denote by $\hat{\alpha}'$ the prefix of $\hat{\alpha}$ that corresponds to α'. In the rest of this section we will be using this notation in comparing the communication efficiency of the two algorithms in terms of gossip messaging.

Recall that gossip messages in Rambo have the format $\langle W, v, t, cm, pns, pnr\rangle$, and in LL-Rambo the format is $\langle W, D, v, t, cm, pns, pnr\rangle$. We also assume that each node identifiers is γ bits long, and that the values v, t, cm, pns, and pnr, altogether occupy δ bits (a constant) in gossip messages.

Scenario 1: No joins or departures after α'. Here we make the following additional assumptions about α' for LL-Rambo and $\hat{\alpha}'$ for Rambo. Consider the following sequence of events, identical for α' and $\hat{\alpha}'$. The service is initialized by some node, called the *creator*. Next m new participants join the service by sending to the creator m join requests. After the last join request is received, the creator sends a gossip message to each new participant. Once these gossip messages are received all nodes have active status. At this point the cardinality of *world* for each node is $n = m + 1$. Now, l nodes decide to leave the system. In α' of LL-Rambo these nodes send leave notification messages to all participating nodes. We assume that these messages are delivered. These notification messages are of constant size and are much smaller than any gossip message. Now the cardinality of the *departed* set at each node is l in LL-Rambo. In the case of Rambo, l nodes leave by emulating crash failures. The number of the active nodes, a, is $a = n - l$. This concludes α' (respectively $\hat{\alpha}'$), following which no new nodes join the system or no active nodes leave the system. Following α' (respectively $\hat{\alpha}'$), normal timing holds in α (respectively $\hat{\alpha}$), and the active nodes gossip at regular intervals as described above. We now give the result that compares the two algorithms for r rounds of gossip.

Proposition 5.2: Let α be α'-*normal* execution of LL-Rambo and $\hat{\alpha}$ be $\hat{\alpha}'$-*normal* execution of Rambo as defined by Scenario 1. Then:

(a) there are $r \cdot a \cdot l$ fewer gossip messages in α following α' than in $\hat{\alpha}$ following $\hat{\alpha}'$;

(b) the bit complexity of gossip messages in α following α' is smaller by $((r-1)\cdot n^2 \cdot a \cdot \gamma) + (a \cdot l^2 \cdot \gamma) + (r \cdot a \cdot l \cdot \delta)$ than the bit complexity in $\hat{\alpha}$ following $\hat{\alpha}'$.

Proof. Nodes participating in Rambo can not distinguish nodes that failed from nodes that departed. Therefore, in each round of gossip Rambo sends $a \cdot n$ messages (an active node sends gossip messages to all nodes in its *world*). Each gossip message has the size of $n \cdot \gamma + \delta$ bits ($|W| = |world| = n$). Hence, the gossip message complexity of Rambo for r rounds is $r \cdot a \cdot n$, and the gossip message bit complexity is $r \cdot a \cdot n \cdot (n \cdot \gamma + \delta)$.

In LL-Rambo each node has learned that l nodes departed. Therefore, a^2 gossip messages are sent in each of the r rounds. In the first round, each gossip message has size $(n + l) \cdot \gamma + \delta$ (since nodes do not know what other nodes know). Since all messages are delivered, at the end of the first round each node knows that every active node has full knowledge of *world* and *departed* sets. Hence, in the following rounds the gossip message size is δ bits. Therefore, the gossip message complexity of LL-Rambo for r rounds is $r \cdot a^2$, and the gossip message bit complexity is $\big(a^2 \cdot (n+l) \cdot \gamma\big) + \big(r \cdot a^2 \cdot \delta\big)$.

Therefore, LL-Rambo sends $r \cdot a \cdot l$ fewer gossip messages than Rambo. The reduction for the gossip message bit complexity is $\big((r-1) \cdot n^2 \cdot a \cdot \gamma\big) + \big(a \cdot l^2 \cdot \gamma\big) + (r \cdot a \cdot l \cdot \delta)$. □

If l is zero then the behavior of these two algorithms is identical. Otherwise, l can be as small as one and as large as $n-1$, and since $a = n - l$, the savings in gossip messages for LL-Rambo are between $\Omega(r \cdot n)$ and $O(r \cdot n^2)$. The reduction in bit complexity for LL-Rambo is between $\Omega(r \cdot n^2)$ and $O(r \cdot n^3)$ (γ and δ are assumed to be constants).

Scenario 2: Steady turnover rate after α'. In this scenario, following the initial segment of an execution, nodes join and leave at a constant rate per round in both Rambo and LL-Rambo. Consider the following sequence of events for both systems. As in Scenario 1, the service is initialized by the creator. Then m new participants join the service by sending join requests to the creator. After the last join request is received, the creator sends gossip to all participants. Once these gossip messages are received all nodes are active. At this point the cardinality of *world* at each node is $n = m + 1$. This concludes α' of Rambo (respectively $\hat{\alpha}'$ of LL-Rambo). Following α' (respectively $\hat{\alpha}'$), normal timing holds in α (respectively $\hat{\alpha}$), and the active nodes gossip at the beginning of each round. In this scenario nodes join and leave the service at the same rate: during each round following α' (respectively $\hat{\alpha}'$), z new nodes join and z active nodes leave the service. We now give the result that compares the two algorithms for r rounds of gossip.

Proposition 5.3: Let α be α'-*normal* execution of LL-Rambo and $\hat{\alpha}$ be $\hat{\alpha}'$-*normal* execution of Rambo as defined by Scenario 2. Then:

(a) there are $(n-z) \cdot z \cdot ((r-1) \cdot (r-2)/2)$ fewer gossip messages in α following α' than in $\hat{\alpha}$ following $\hat{\alpha}'$;

(b) the bit complexity of gossip messages in α following α' for $r > 2$ is smaller by

$$(n-z)\cdot\Big(\gamma\cdot z^2\cdot\big((r-1)\cdot(r-2)\cdot(2\cdot r-1)/6\big)+\big(2\cdot\gamma\cdot n+\delta\big)\cdot z\cdot\big((r-1)\cdot(r-2)/2\big)+n\cdot\gamma\cdot(n-z)\cdot(r-2)+n^2\cdot\gamma\Big)$$

than the bit complexity in $\hat{\alpha}$ following $\hat{\alpha}'$. For $r = 2$ the reduction is $(n-z) \cdot n^2 \cdot \gamma$ and for $r = 1$ the bit complexity is the same.

Proof. For the analysis we assume that the join requests are received by all active nodes by the end of the round in which they were sent, and are receive after all gossip messages are sent. On the other hand, we assume that the leave notifications are received (in LL-Rambo) by active nodes at the end of the following round. These assumptions yield worst case behavior for LL-Rambo in Scenario 2 since delaying notification messages only affect the gossip communication complexity of LL-Rambo (in Rambo no notification messages are sent).

We now analyze the behavior of Rambo for a round $1 \le k \le r$. By the code of Rambo, we observe that the *world* set grows as nodes join the system. Since gossip messages include the entire *world* set it follows that the size of gossip messages is proportional to the size of *world*. In Rambo nodes leave by emulating failures. Since nodes can not detect such failures, each active node attempts to gossip to all members in its *world*, regardless of their failure status. Hence, as new participants join, the number and the size of gossip messages sent in Rambo increases.

We separately consider the case when $k = 1$. At the beginning of the first round the *world* of each active node is n (join requests have not been yet received). From this and the fact that only active nodes gossip (new joined nodes become active after they receive a gossip message), in first round n^2 messages are sent each of size $n \cdot \gamma + \delta$. We now consider $2 \le k \le r$. At the beginning of the k'th round there are $n - z$ active nodes. Observe that up to k'th round $(k-1) \cdot z$ active nodes left the system, also the same number attempted to join the system, of which only $(k-2) \cdot z$ became active by k'th round (it takes at most $2d$ time for a node to become active). Hence, there may be at most up to $n - z$ active nodes and n participating nodes in the system at the beginning of each round k. The *world* set of each active node has cardinality $n + (k-1) \cdot z$ (since z nodes enter the system in every round). Therefore, in each round $k > 1$, $(n-z) \cdot (n + (k-1) \cdot z)$ gossip messages are sent, each of size $(n + (k-1) \cdot z) \cdot \gamma + \delta$.

We now analyze, in a similar fashion, the behavior of LL-Rambo for a round $1 \leq k \leq r$. By the code of LL-Rambo we observe that both the *world* and the *departed* sets grow as the new participants join and active nodes leave the system. However, two nodes that have been active in two consecutive rounds, need only to exchange gossip messages that contain only the identifiers of the z nodes that have departed and the z nodes that have joined the system in the previous round. On the other hand, an active node has to send all current knowledge about *world* and *departed* to the newly joined nodes. Also note that once a node learns that another node departed from the system it does not send any further gossip messages to that node. We separately consider cases where $k = 1$ and $k = 2$. The argument for the first round is identical to the argument of the first round of Rambo. At the beginning of the second round there are $n - z$ active nodes (this is because in the first round z nodes departed and the z nodes joined the system, where the newly joined nodes are not active yet). Also the cardinality of *world* variable of each active node is $n + z$ and the cardinality of *departed* variable is zero (since the leave notifications have not been received yet – will be received at the end of this round). Therefore, in the second round, $(n - z) \cdot (n + z)$ gossip messages are sent, and the total bit complexity of these messages is $(n-z)\cdot\Big(n\cdot(z\cdot\gamma+\delta)+z\cdot((n+z)\cdot\gamma+\delta)\Big)$. We now consider $2 \leq k \leq r$. At the beginning of the round k there are $n - z$ active nodes (using the same reasoning as in case of Rambo for rounds $k \geq 2$). The *world* set of each active node has cardinality $n + (k - 1) \cdot z$ (since z nodes enter the system in every round) and the *departed* set has cardinality $(k - 2) \cdot z$ (since it takes an extra round for the notifications to be received). It follows that in round $k > 2$, $(n - z) \cdot (n + z)$ gossip messages are sent, and the total bit complexity of these messages is $(n-z)\cdot\Big(n\cdot(2\cdot z\cdot\gamma+\delta)+z\cdot((n+(k-1)\cdot z+(k-2)\cdot z)\cdot\gamma+\delta)\Big)$. By performing elementary mathematical computations, it follows that for r rounds, LL-Rambo sends $(n - z) \cdot z \cdot ((r - 1) \cdot (r - 2)/2)$ less gossip messages than Rambo. Also, it is not difficult to see that in the first round LL-Rambo gossip bit complexity is the same as Rambo, and the gossip bit complexity reduction for the second round is $(n-z)\cdot n^2\cdot\gamma$. Finally, for $r > 2$ we have that the savings in the gossip bit complexity are $(n-z)\cdot\Big(\gamma\cdot z^2\cdot\big((r-1)\cdot(r-2)\cdot(2\cdot r-1)/6\big)+\big(2\cdot\gamma\cdot n+\delta\big)\cdot z\cdot\big((r-1)\cdot(r-2)/2\big)+n\cdot\gamma\cdot(n-z)\cdot(r-2)+n^2\cdot\gamma\Big)$. $\square$

For the following bounds we consider $1 \leq z \leq n - 1$. The savings in gossip messages for LL-Rambo when $r > 2$ are between $\Omega(r^2 \cdot n)$ and $O(r^2 \cdot n^2)$. When $1 \leq r \leq 2$ there are no savings. The reduction in gossip bit complexity for LL-Rambo when $r > 1$ is between $\Omega(n^2)$ and $O(r^3 \cdot n^3)$ (γ and δ are assumed to be constants). When $r = 1$ there is no reduction.

6 LL-Rambo Implementation

We developed proof-of-concept implementations of Rambo and LL-Rambo on a network-of-workstations. In this section we presents preliminary experimental results.

Experimental Results. We developed the system 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 [3] that guided the derivation of Java code. The platform consists of a Beowulf cluster with ten machines running Linux. The machines are various Pentium processors up to 900 MHz interconnected via a 100 Mbps Ethernet switch. The implementation of the two algorithms share most of the code and all low-level routines, so that any difference in performance is traceable to the distinct *world* and *departed* set management and the gossiping discipline encapsulated in each algorithm.

We are interested in long-lived applications and we assume that the number of participants grows arbitrarily. Given the limited number of physical nodes, we use majority quorums of the these nodes, and we simulate a large number of other nodes that join the system by including such node identifiers in the *world* sets. Using non-existent nodes approximates the behavior of a long-lived system with a large set of participants. However, when using all-to-all gossip that grows quadratically in the number of participants, it is expected that the differences in Rambo and LL-Rambo performance will become more substantial when using a larger number of physical nodes.

The experiment is designed as follows. There are ten nodes that do not leave the system. These nodes perform concurrent read and write operations using a single configuration (that does not change over time), consisting of majorities, i.e., six nodes. Figure 4 compares (a) the average latency of gossip messages and

(b) the average latency of read and write operations in RAMBO and LL-RAMBO, as the cardinality of *world* sets grows from 10 to 7010.

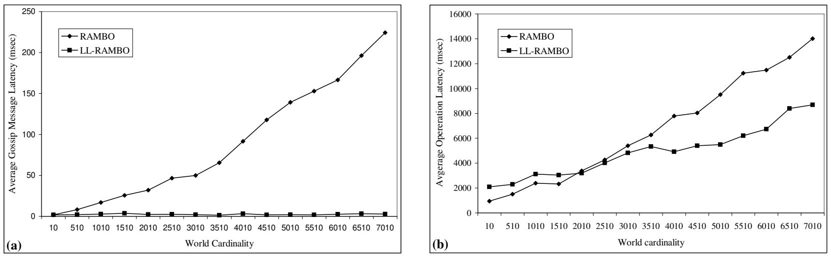

Fig. 4: Preliminary empirical results: (a) gossip message latency, (b) read and write latency.

LL-RAMBO exhibits substantially better gossip message latency than RAMBO (Fig. 4(a)). In fact the average gossip latency in LL-RAMBO does not vary noticeably. On the other hand, the gossip latency in RAMBO grows substantially as the cardinality of the *world* sets increases. This is expected due to the smaller incremental gossip messages of LL-RAMBO, while in RAMBO, the size of the gossip messages is always proportional to the cardinality of the *world* set. LL-RAMBO trades local resources (computation and memory) for smaller and fewer gossip messages. We observe that the read/write operation latency is slightly lower for RAMBO when the cardinality of the *world* sets is small (Fig. 4(b)). As the size of the *world* sets grows, the operation latency in LL-RAMBO becomes substantially better than in RAMBO.

We close this section with the following remarks. First, the experiment is designed using a single configuration that does not change over time. Note that reconfiguration creates additional network traffic, however, when reconfigurations are infrequent performance of the implementation is not significantly affected. Since the main aim of our algorithm is to reduce the size of the gossip messages we decided not to reconfigure during the experiment. Future experiments will include reconfiguration activities, however it is also important to evaluate the system when the reconfiguration traffic does not interfere with the routine gossip.

Second, the experiments were conducted using our preliminary implementation. This implementation is a proof-of-concept for the methodology developed in [3] to translate the IOA specification into Java code. Both RAMBO and LL-RAMBO have been implemented faithfully to their specification, however no attempt was made to optimize either system. Therefore, the performance improvements in Fig(s). 4(a) and (b) should be considered in relative terms.

Finally, observe that due to node failures some gossip messages may continue to grow without bound (since sender does not receive acknowledgments from the failed node). To diminish communication and processing impact in this case we use *exponential backoff*: if a node does not receive a gossip message from some other node it will double the delay between consecutive gossip rounds to that node; normal timing is restored once communication is reestablished. Note that both algorithms may take advantage of this assumption. In our experiments, gossip messages were not sent to the virtual-failed nodes (this approximates exponential backoff in the case of permanently crashed nodes). The results in Fig(s). 4(a) and (b) illustrate performance of RAMBO and LL-RAMBO when gossip messages are exchanged only between non-failed nodes.

7 Discussion and Future Work

We presented an algorithm for long-lived atomic data in dynamic networks. Prior solutions for dynamic networks [1, 2] did not allow the participants to leave gracefully and relied on gossip that involved sending messages whose size grew with time. The new algorithm, called LL-RAMBO improves on prior work by supporting graceful departures of participants and implementing incremental gossip. The algorithm substantially reduces the size and the number of gossip messages, leading to improved performance of the read and write operations. We show that the improved algorithm implements atomic objects in the presence of arbitrary asynchrony, dynamic node joins and departures. The algorithm relies on simple point-to-point channels that permit message loss and reordering.

We analyzed the efficiency of the algorithm and illustrated its performance using our implementation of the algorithm in the local-area setting. In trading knowledge for communication, the algorithm increases the local memory requirements. Specifically, the needed local storage is quadratic in the number of active participants. Future plans include exploring optimizations that will reduce the local storage usage to be about linear in the number of participants. Also, we plan to further explore ways to decrease the number of gossip messages sent. Currently, LL-RAMBO allows all-to-all exchange of information among active participants, and when gossip is unconstrained and communication bandwidth is limited, network congestion may degrade the performance of the system. One way to solve this problem is to constrain the gossip patterns. Restricting gossip vacuously preserves the linearizability property of the algorithm, but will possibly alter its performance.

Acknowledgements. The authors thank Nancy Lynch and Seth Gilbert for many discussions.

References

[1] N. Lynch and A.A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. of 16th International Symposium on Distributed Computing*, pages 173–190, 2002.

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

[3] P.M. Musial and A.A. Shvartsman. Implementing a reconfigurable atomic memory service for dynamic networks. In *Proc. of 18'th International Parallel and Distributed Symposium — FTPDS WS*, page 208b, 2004.

[4] N.A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical report, 1987.

[5] D.K. Gifford. Weighted voting for replicated data. In *Proc. of 7th ACM Symp. on Oper. Sys. Princ.*, pages 150–162, 1979.

[6] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Sys.*, 4(2):180–209, 1979.

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

[8] B. Awerbuch and P. Vitanyi. Atomic shared register access by asynchronous hardware. In *Proc. of 27th IEEE Symposium on Foundations of Computer Science*, pages 233–243, 1986.

[9] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *J. of the ACM*, 42(1):124–142, 1996.

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

[11] N. Lynch and A.A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proc. of 27th Int-l Symp. on Fault-Tolerant Comp.*, pages 272–281, 1997.

[12] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of the 11th ACM Symposium on Operating Systems Principles*, December 1987.

[13] Special issue on group communication services. *Communications of the ACM*, 39(4), 1996.

[14] A. El Abbadi, F. Cristian, and D. Skeen. An efficient fault-tolerant protocol for replicated data management. In *Proc. of 4th ACM Symposium on Principles of Databases*, pages 215–228, 1985.

[15] R. De Prisco, A. Fekete, N. Lynch, and A.A. Shvartsman. A dynamic primary configuration group communication service. In *Proc. of 13th Int-l Conference on Distributed Computing*, pages 64–78, 1999.

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

[17] D. Dolev, E. Lotem, and I. Keidar. Dynamic voting for consistent primary components. In *Proc. of 16th ACM Symposium on Principles of Distributed Computing*, pages 63–71, 1997.

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

[19] K.P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. on Computer Systems*, 17(2):41–88, 1999.

[20] O.M. Cheiner and A.A. Shvartsman. Implementing an eventually-serializable data service as a distributed system building block. volume 45, pages 43–71, 1999.

[21] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of 6th ACM Symp. on Princ. of Distr. Comp.*, pages 1–12, 1987.

[22] A. Ghorbani and V. Bhavsar. Training artificial neural networks using variable precision incremental communication. In *Proc. of IEEE World Congress On Computational Intelligence*, volume 3, pages 1409–1414, 1994.

[23] K. Petersen, M.J. Spreitzer, D.B. Terry, and M.M. Theimer. Bayou: Replicated database services for world-wide applications. In *Proc. of 7th ACM SIGOPS European Workshop*, pages 275–280, 1996.

[24] Y. Minsky. *Spreading rumors cheaply, quickly, and reliably.* PhD thesis, Cornell University, 2002.

[25] R. Van Renesse, K.P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. on Computer Systems*, 21(2):164–206, 2003.

[26] R.A. Golding. *Weak-consistency group communication and membership.* PhD thesis, University of California, 1992.

[27] M. Rabinovich, N. Gehani, and A. Kononov. Efficient update propagation in epidemic replicated databases. In *Proc. of 5th Int. Conf. on Extending Database Technology*, pages 207–222, 1996.

[28] R.G. Guy, J.S. Heidemann, W. Mak, T.W. Page Jr., G.J. Popek, and D. Rothmeier. Implementation of the ficus replicated file system. In *Proc. of Summer USENIX Conference*, pages 63–71, 1990.

[29] C. Georgiou, P.M. Musial, and A.A. Shvartsman. Long-lived RAMBO: Trading knowledge for communication. In *Proc. of 11'th Colloquium on Structural Information and Communication Complexity*, pages 185–196. Springer, 2004.

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

[31] Sh. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. Welch. Geoquorums: Implementing atomic memory in ad hoc networks. In *Proc. of 17th International Symposium on Distributed Computing*, pages 306–320, 2003.

[32] N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

APPENDIX

A RAMBO Specification

Signature:

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

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

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

Transitions:

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

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

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

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

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

Input fail_i
Effect:
  $failed \leftarrow \mathsf{true}$

Fig. 5: $Joiner_i$: Signature, state, and transitions

Signature:

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

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

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

State:

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

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
    $target \in \mathbb{N}$

Fig. 6: $Reader\text{-}Writer_i$: Signature and state

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

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

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

Fig. 7: $Reader\text{-}Writer_i$: Join-related transitions

Input $\mathsf{cfg\text{-}upgrade}(k)_i$
Effect:
  $\neg failed$
  $status = \mathsf{active}$
  $upg.phase = \mathsf{idle}$
  $cmap(k) \in C$
  $\forall \ell \in \mathbb{N}, \ell < k : cmap(\ell) \neq \perp$

$pnum1 \leftarrow pnum1 + 1$
$upg \leftarrow \langle \mathsf{query}, pnum1, cmap, \emptyset, k \rangle$

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

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

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:
  $upg.phase \leftarrow \mathsf{idle}$

Fig. 8: $Reader\text{-}Writer_i$: Configuration-Management transitions

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 $\mathsf{recv}(\langle W, v, t, cm, pns, pnr, m\rangle)_{j,i}$
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
      $pnum1 \leftarrow pnum1 + 1$
      $op.acc \leftarrow \emptyset$
      $op.cmap \leftarrow truncate(cmap)$
    if $upg.phase \neq \mathsf{idle}$ and $pnr \geq upg.pnum$ then
     $upg.acc \leftarrow upg.acc \cup \{j\}$

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

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

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

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

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

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

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

Input fail_i
Effect:
  $failed \leftarrow \mathsf{true}$

Fig. 9: $Reader\text{-}Writer_i$: Read/write and failure transitions

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

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

Fig. 10: $Cons(k, c)$: External signature ($Cons(k, c)$ is an external consensus service)

Signature:

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

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

State:

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

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

Fig. 11: $Recon_i$: Signature and state

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

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

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

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

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

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

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

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

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

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

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

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

Input fail_i
Effect:
  $\mathit{failed} \leftarrow \mathsf{true}$

Fig. 12: Recon_i: Transitions.