Long-Lived Rambo: Trading Knowledge for Communication*

Chryssis Georgiou[1], Peter M. Musial[2], and Alexander A. Shvartsman[2,3]

[1] Department of Computer Science, University of Cyprus, Nicosia, Cyprus
[2] Department of Computer Science & Engineering, University of Connecticut, Storrs, CT, USA
[3] CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

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 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 only way to ensure survivability of data is 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 [10] and extended by Gilbert *et al.* [6]. This memory service, called RAMBO (Reconfigurable Atomic Memory for Basic Objects) maintains atomic (linearizable) readable/writable data in highly dynamic environments. In order to achieve availability in the presence of failures, the objects are replicated at

* This work is supported in part by the NSF Grants 9984778, 9988304, 0121277, and 0311368.

several 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-* 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. 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 failures, and message loss. Atomicity is guaranteed in any execution of the algorithm [10, 6].

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. The algorithm propagates information among the participants by means of gossip messages that contain information corresponding to the sender's state. The number and the size of gossip message 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 [6, 10], we use Input/Output Automata [11] 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. 5).

Background. Several approaches can be used to implement consistent data in (static) distributed systems. Many algorithms used collections of intersecting sets of object replicas to solve consistency problems, e.g., [2, 14, 15]. Extension with reconfigurable quorums have been explored [4], but this system has limited ability to support long-lived data when the longevity of processors is limited. Virtual synchrony [3], and group communication services (GCS) in general [1], 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.

The work on reconfigurable atomic memory [4, 10, 6] 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. Here we provide a long-lived solution 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., [7, 12]. Incremental gossip is also called anti-entropy [5, 13] or reconciliation [8]; 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.

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. Section 5 gives the analysis and experimental results. (Complete proofs and analysis are found in MIT/LCS/TR-943.)

2 Reconfigurable Atomic Memory for Basic Objects (RAMBO)

We now describe the RAMBO algorithm as presented in [10], including the rapid configuration upgrade as given in [6]. 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 [10, 6]. 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.

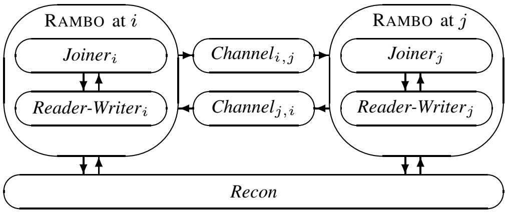

Fig. 1. RAMBO architecture depicting automata at nodes i and j, the channels, and the *Recon* service.

The algorithm consists of three kinds of automata: (i) *Joiner* automata, handling join requests, (ii) *Recon* automata, handling reconfiguration requests 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 simply sends a join message when node i joins, and sends a join-ack message whenever a join message is received. The *Recon* automaton establishes a total ordering of configurations (for details see [10]).

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.

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

Input:	$\mathsf{join}(\mathsf{rambo}, J)_i,\ J \subseteq I - \{i\}, i \in I$, such that if $i = i_0$ then $J = \emptyset$	**Output:**	$\mathsf{join\text{-}ack}(\mathsf{rambo})_i, i \in I$
	$\mathsf{read}_i,\ i \in I$		$\mathsf{read\text{-}ack}(v)_i, v \in V, i \in I$
	$\mathsf{write}(v)_i,\ v \in V, i \in I$		$\mathsf{write\text{-}ack}_i, i \in I$
	$\mathsf{recon}(c, c')_i,\ c, c' \in C, i \in members(c), i \in I$		$\mathsf{recon\text{-}ack}(b)_i, b \in \{ok, nok\}, i \in I$
	$\mathsf{fail}_i,\ i \in I$		$\mathsf{report}(c)_i, c \in C, i \in I$

Fig. 2. RAMBO: External signature.

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 has 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 tracks 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 records which nodes from which quorums have responded during the current phase.

Finally, the nodes communicate via asynchronous unreliable point-to-point channels. 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

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

State:
As in RAMBO, plus new states:
$leave\text{-}world$, a finite subset of $I$, initially $\emptyset$
$departed$, a finite subset of $I$, initially $\emptyset$
$\boxed{\begin{array}{l} ig \in IGMap, \text{ initially } \forall k \in I, \\ ig(k).wk = \emptyset,\ ig(k).w\text{-}ua = \emptyset, \\ ig(k).dk = \emptyset,\ ig(k).d\text{-}ua = \emptyset, \\ ig(k).p\text{-}ack = 0 \end{array}}$

Transitions at $i$:
Input $\mathsf{recv}(\langle W, D, v, t, cm, pns, pnr\rangle)_{j,i}$
Effect:
  if $\neg failed \wedge status \neq$ idle then
    $status \leftarrow$ active
    $world \leftarrow world \cup W$
    $departed \leftarrow departed \cup D$
    $\boxed{\begin{array}{l} [h]hr\text{-}W(j, i, pnr) \leftarrow W \\ [h]hr\text{-}D(j, i, pnr) \leftarrow D \\ ig(j).wk \leftarrow ig(j).wk \cup W \\ ig(j).w\text{-}ua \leftarrow ig(j).w\text{-}ua - W \\ ig(j).dk \leftarrow ig(j).dk \cup D \\ ig(j).d\text{-}ua \leftarrow ig(j).d\text{-}ua - D \\ \text{if } pnr > ig(j).p\text{-}ack \text{ then} \\ \quad ig(j).wk \leftarrow ig(j).wk \cup ig(j).w\text{-}ua \\ \quad ig(j).w\text{-}ua \leftarrow world - ig(j).wk \\ \quad ig(j).dk \leftarrow ig(j).dk \cup ig(j).d\text{-}ua \\ \quad ig(j).d\text{-}ua \leftarrow departed - ig(j).dk \\ \quad ig(j).p\text{-}ack \leftarrow pnum1 \end{array}}$
    if $t > tag$ then $(value, tag) \leftarrow (v, t)$
    $cmap \leftarrow update(cmap, cm)$
    $pnum2(j) \leftarrow \max(pnum2(j), pns)$
    if $op.phase \in \{$query, prop$\} \wedge pnr \geq op.pnum$ then
      $op.cmap \leftarrow extend(op.cmap, truncate(cm))$
      if $op.cmap \in Truncated$ then
        $op.acc \leftarrow op.acc \cup \{j\}$
      else
        $op.acc \leftarrow \emptyset$
        $op.cmap \leftarrow truncate(cmap)$
    if $upg.phase \in \{$query, prop$\} \wedge pnr \geq upg.pnum$ then
      $upg.acc \leftarrow upg.acc \cup \{j\}$

input $\mathsf{recv(leave)}_{j,i}$
Effect:
  if $\neg failed \wedge status =$ active then
   $departed \leftarrow departed \cup \{j\}$

Output $\mathsf{send}(\langle W, D, v, t, cm, pns, pnr\rangle)_{i,j}$
Precondition:
   $\neg failed$
   $status =$ active
   $j \in (world - departed)$
   $\langle W, D, v, t, cm, pns, pnr\rangle =$
     $\langle world \boxed{-ig(j).wk}, departed \boxed{-ig(j).dk},$
       $value, tag, cmap, pnum1, pnum2(j) \rangle$
Effect:
   $pnum1 \leftarrow pnum1 + 1$
   $[h]h\text{-}msg \leftarrow h\text{-}msg \cup$
       $\langle\langle W, D, v, t, cm, pns, pnr\rangle, i, j\rangle$
   $\boxed{\begin{array}{l} [h]hs\text{-}world(i, j, pns) \leftarrow world \\ [h]hs\text{-}departed(i, j, pns) \leftarrow departed \\ [h]hs\text{-}wk(i, j, pns) \leftarrow ig(j).wk \\ [h]hs\text{-}dk(i, j, pns) \leftarrow ig(j).dk \\ [h]hs\text{-}wua(i, j, pns) \leftarrow ig(j).w\text{-}ua \\ [h]hs\text{-}dua(i, j, pns) \leftarrow ig(j).d\text{-}ua \\ [h]hs\text{-}pack(i, j, pns) \leftarrow ig(j).p\text{-}ack \end{array}}$

input $\mathsf{leave}_i$
Effect:
   if $\neg failed$ then
    $failed \leftarrow$ true
    $departed \leftarrow departed \cup \{i\}$
    $leave\text{-}world \leftarrow world - departed$

output $\mathsf{send(leave)}_{i,j}$
Precondition:
   $j \in leave\text{-}world$
Effect:
   $leave\text{-}world \leftarrow leave\text{-}world - \{j\}$
```

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

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-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 Sect. 4 (we combine the two specifications in the interest of space).

The signature of *Reader-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-Writer*$_i$ is extended with new state variables: $departed_i$, the set of nodes that left the system, as known at node i, $\mathit{leave\text{-}world}_i$, the set of nodes that node i can inform of its own departure, once it decides to leave and sets $\mathit{leave\text{-}world}_i$ to $\mathit{world} - \mathit{departed}$.

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, equal to the $\mathit{departed}$ 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-Writer*$_i$. The leave_i action at *Reader-Writer*$_i$ (see Fig. 3) also initializes the set $\mathit{leave\text{-}world}_i$ to the identifiers found in world_i, less those found in $\mathit{departed}_i$. Now *Reader-Writer*$_i$ is allowed to send one leave notification to any node in $\mathit{leave\text{-}world}_i$. This is done by the $\mathsf{send(leave)}_{i,j}$ action that arbitrarily chooses the destination j from $\mathit{leave\text{-}world}_i$. Note that node i may nondeterministically choose the original fail_i action, in which case no notification messages are sent (this is the "non-graceful" departure).

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

Atomicity of L-RAMBO *service.* The L-RAMBO system is the composition of all *Reader-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 $\mathit{h\text{-}msg}_i$ to be the history variable that keeps track of all messages sent by *Reader-Writer*$_i$ automata. Initially, $\mathit{h\text{-}msg}_i = \emptyset$ 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 $\mathit{h\text{-}msg}_i \leftarrow \mathit{h\text{-}msg}_i \cup \{\langle m, i, j\rangle\}$. We define *h-MSG* to be $\bigcup_{i \in I} \mathit{h\text{-}msg}_i$. (The remaining history variables are used in reasoning about LL-RAMBO, see Sect. 4).

The following lemma states that only good messages are sent.

Lemma 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-MSG, and $m = \langle W, D, v, t, cm, pns, pnr\rangle$ or $m = \mathsf{leave}$ or $m = \mathsf{join}$.*

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 relation [9] from L-RAMBO to RAMBO.

Theorem 1. L-RAMBO *implements atomic read/write objects.*

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 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, 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 $\langle wk, dk, w\text{-}ua, d\text{-}ua, p\text{-}ack \rangle$, where wk, dk, $w\text{-}ua$, and $d\text{-}ua$ are finite subsets of I, and $p\text{-}ack$ 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-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 $j \in world_i$, $ig(j)_i.wk$ is the set of node identifiers that i is assured is a subset of $world_j$, $ig(j)_i.w\text{-}ua$ is the set of node identifiers, a subset of $world_i$, that j needs to acknowledge. The components $ig(j)_i.dk$ and $ig(j)_i.d\text{-}ua$ 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 $p\text{-}ack$ 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 *late* messages. (This is discussed later.)

The lines annotated with $[h]$ in Fig. 3 deal with history variables that are used only in the proof of correctness.

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, $ig(j)_i.wk$, 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 $world_j$, $ig(i)_j.wk$, and $ig(i)_j.w\text{-}ua$. The update of $world_j$ is identical to that in L-RAMBO. By definition $ig(i)_j.wk$ is the set of node identifiers that j is assured that i has, hence we update it with information in W. Similarly, by definition $ig(i)_j.w\text{-}ua$ 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 $ig(i)_j.w\text{-}ua$ from $ig(i)_j.w\text{-}ua$; 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 late message since there must have been a prior message from i to j that included this or higher 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 $ig(i)_j.w\text{-}ua$ 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 $ig(i)_j.w\text{-}ua$ and to add it to $ig(i)_j.wk$.

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 $hs\text{-}W(i, j, p)$ 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 $hs\text{-}D(i, j, p)$, 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 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 $departed_i$, $hs\text{-}wk(i, j, pns)$ records the value of $ig(j)_i.wk$, $hs\text{-}dk(i, j, pns)$ records the value of $ig(j)_i.dk$, $hs\text{-}wua(i, j, pns)$ records the value of $ig(j)_i.w\text{-}ua$, and $hs\text{-}dua(i, j, pns)$ records the value of $ig(j)_i.d\text{-}ua$.
- $hs\text{-}pack(i, j, pns)$ 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 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 $2^I \cup \{\perp\}$. $hr\text{-}W(j, i, pns)$ records the component W (*world*) and $hr\text{-}D(j, i, pns)$ records the component D (*departed*).

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

Lemma 2. *Consider a step* $\langle s, e, s' \rangle$ *of an execution* α *of* LL-RAMBO*, where* $e = \mathsf{recv}(\langle W, D, v, t, cm, p_j, p_i \rangle)_{j,i}$ *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) $s.ig(j)_i.w\text{-}ua \subseteq s.hs\text{-}wua(i, j, p_i)$,
(c) $s.ig(j)_i.d\text{-}ua \subseteq s.hs\text{-}dua(i, j, p_i)$.

Invariant 1 is used in proving the correctness of LL-RAMBO. In Invariant 1, parts (a) to (e) and Lemma 1 are used to show the key parts (f) and (g).

Invariant 1 *For all states* s *of any execution* α *of* LL-RAMBO*:*
(a) $\langle \langle W, D, v, t, cm, pns, pnr \rangle, i, j \rangle \in s.h\text{-}MSG \Rightarrow W \subseteq s.world_i \wedge D \subseteq s.departed_i$,
(b) $\forall\, i, j \in I : s.ig(j)_i.w\text{-}ua \subseteq s.world_i - s.ig(j)_i.wk$,
(c) $\forall\, i, j \in I : s.ig(j)_i.d\text{-}ua \subseteq s.world_i - s.ig(j)_i.dk$,
(d) $\langle \langle W, D, v, t, cm, p, pnr \rangle, i, j \rangle \in s.h\text{-}MSG \Rightarrow s.hs\text{-}wua(i, j, p) \subseteq W$,
(e) $\langle \langle W, D, v, t, cm, p, pnr \rangle, i, j \rangle \in s.h\text{-}MSG \Rightarrow s.hs\text{-}dua(i, j, p) \subseteq D$,
(f) $\forall\, i, j \in I : s.ig(j)_i.wk \subseteq s.world_j$*, and*
(g) $\forall\, i, j \in I : s.ig(j)_i.dk \subseteq s.departed_j$.

Parts (f) and (g) of Invariant 1 show that no node overestimates the knowledge of another node about its *world* and *departed* sets. 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 2. LL-RAMBO *implements atomic read/write objects.*

Proof. (Sketch) We define a 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 (e.g., for $i \in I$, $t.world_i = s.world_i$, $t.pnum1_i = s.pnum1_i$, etc.) and (b) a message $m = \langle W, D, v, t, cm, pns, pnr \rangle$ in the *Channel* automaton of LL-RAMBO to a message $m' = \langle W, D, v, t, cm, pns, pnr \rangle$ in the *Channel* automaton of L-RAMBO so that: $m.v = m'.v$, $m.t = m'.t$, $m.cm = m'.cm$, $m.pns = m'.pns$, $m.pnr = m'.pnr$, $m.W = hs\text{-}world(i, j, pns) - hs\text{-}wk(i, j, pns)$ and $m'.W = hs\text{-}world(i, j, pns)$, and $m.D = hs\text{-}departed(i, j, pns) - hs\text{-}dk(i, j, pns)$ and $m'.D = hs\text{-}departed(i, j, pns)$. Using the specifications of the two algorithms and Invariant 1, we show that R is a simulation mapping from LL-RAMBO to L-RAMBO. Since L-RAMBO implements atomic objects per Theorem 1, so does LL-RAMBO.

5 LL-RAMBO Implementation and Performance

We developed proof-of-concept implementations of RAMBO and LL-RAMBO on a network-of-workstations. In this section we presents preliminary experimental results and overview conditional analysis of algorithms.

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 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.

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.

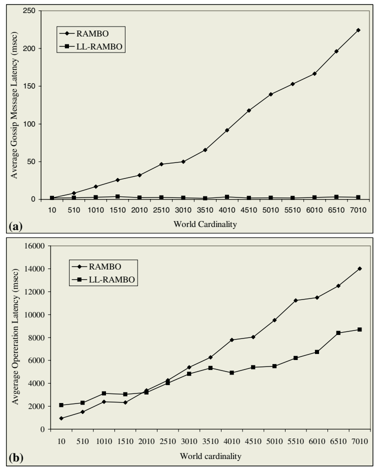

Fig. 4. Preliminary empirical results: (a) gossip message latency, (b) read and write latency.

Performance analysis. We briefly summarize our performance analysis of LL-RAMBO. Here we assume that the participating nodes perpetually gossip with a period d. We show that the latency of read and write operations matches that of RAMBO [10, 6]. Specifically, if d is the maximum message delay, then read and write operations take at most $8d$ time, when reconfigurations are not too frequent. We analyze the communication of LL-RAMBO in the following scenario. We consider the scenario where, once an object is created, several nodes join the system, such that, together with the creator, there are n nodes. Then l nodes leave the system, such that the number of remaining active nodes is $a = n - l$. We show that in this case, after r rounds of gossip, the savings in gossip messages for LL-RAMBO are between $\Omega(r \cdot n)$ and $O(r \cdot n^2)$.

6 Discussion and Future Work

We presented an algorithm for long-lived atomic data in dynamic networks. Prior solutions for dynamic networks [10, 6] 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. Our improvements are formally specified and proved.

Acknowledgements. The work of Chryssis Georgiou was performed at the University of Connecticut. The authors thank Nancy Lynch and Seth Gilbert for many discussions.

References

1. Special Issue on Group Communication Services, vol. 39(4) of Comm. of the ACM, 1996.
2. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing Memory Robustly in Message Passing Systems. Journal of the ACM 42(1):124–142, 1996.
3. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. In Proceedings of the 11th ACM Symposium on Operating Systems Principles, December 1987.
4. Englert, B., Shvartsman, A.A.: Graceful quorum reconfiguration in a robust emulation of shared memory. In Proc. of Inter. Conf. on Dist.d Computer Systems, pp. 454–463, 2000.
5. Golding, R.A.: Weak-consistency group communication and membership. PhD Thesis, University of California, 1992.
6. Gilbert, S.,Lynch, N., Shvartsman, A.A.: RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In Proc. of Inter. Conf. on Dependable Systems and Networks, pp. 259–268, 2003.
7. Ghorbani, A., Bhavsar, V.: Training artificial neural networks using variable precision incremental communication. In Proc. of IEEE World Congress On Computational Intelligence, 3:1409–1414, 1994.
8. Guy, R.G., Heidemann, J.S., Mak, W., Page Jr., T.W., Popek, G.J., Rothmeier, D.: Implementation of the Ficus Replicated File System. In Proc. of Summer USENIX Conference, pp. 63-71, 1990.
9. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers, 1996.
10. Lynch, N., Shvartsman, A.A.: RAMBO: A reconfigurable atomic memory service for dynamic networks. In Proc. of 16th Inter. Symp. on Dist. Comp., pp. 173–190, 2002.
11. Lynch, N.A., Tuttle, M.: Hierarchical correctness proofs for distributed algorithms. LCS/TR-387, MIT, 1987.
12. Minsky, Y.: Spreading rumors cheaply, quickly, and reliably. Ph.D Thesis, Cornell University, 2002.
13. Rabinovich, M., Gehani, N., Kononov, A.: Efficient update propagation in epidemic replicated databases. In Proc. of 5th Int. Conf. on Extending Database Tech., pp. 207-222, 1996.
14. Upfal, E., Wigderson, A.: How to share memory in a distributed system. Journal of the ACM 34(1):116–127, 1987.
15. Vitanyi, P., Awerbuch, B.: Atomic shared register access by asynchronous hardware. In Proc. of 27th IEEE Symposium on Foundations of Computer Science, pp. 233–243, 1986.